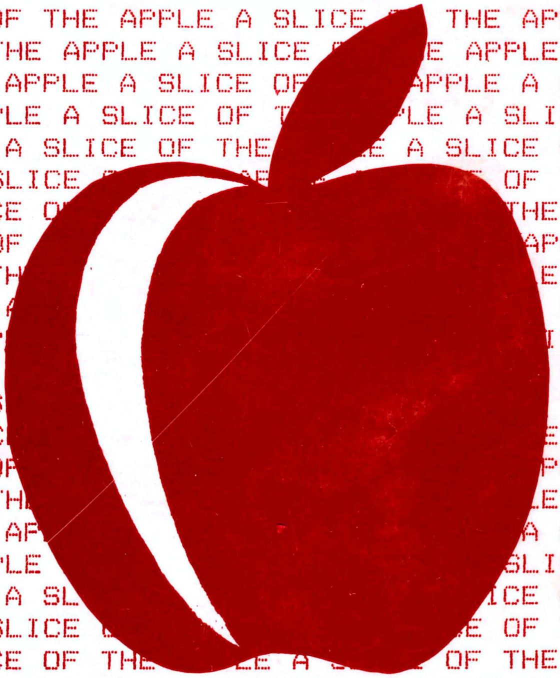
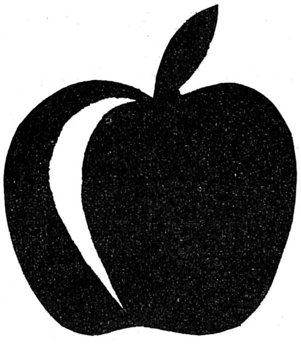
A SLICE OF THE APPLE



A SLICE OF THE APPLE



RONALD L. CULBERTSON
HARRY R. GOLDBERG

1982

©

First Published in 1982

Copyright © 1982

All rights reserved. No part of this book may be reproduced or used in any form or by any means -- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage or retrieval systems -- without written permission of the authors, Harry R. Goldberg and Ronald L. Culbertson.

This book is in no way connected with the Apple Computer Company. Apple, Apple II, Apple II Plus, Disk II, and Applesoft are registered trademarks of Apple Computer Inc. with regard to any computer products.

PREFACE

Over ten years ago a small private school in Baltimore, Maryland began what was a highly innovative attempt at teaching computer programming skills. The attempt was innovative because it was designed for seventh grade math students. Since that time, the program has continued to thrive at Gilman School. Through the school's financial support and its far-sighted curriculum, our program now introduces all seventh graders to twelve full weeks of intensive programming skills. In the sequential math program this seventh grade work is refined and reinforced through the senior level of high school.

This book is the accumulated ten year experiences of teaching computer programming to middle schoolers. We believe firmly, as we did ten years ago, that middle schoolers can and do benefit from the problem-solving skills developed in this work. They have the power to handle sophisticated programming techniques, and this power generates a wave of enthusiasm that is injected into all areas of learning.

A Slice of the Apple is designed specifically for the middle school youngster, though we believe it will also be a valuable aid for any beginning computer student, regardless of age. This book can help direct the efforts of both student and teacher alike in getting a piece of the action in computer programming.

ACKNOWLEDGEMENTS

No book such as this can be of two individuals. This text is a product of efforts on the part of many. The authors wish to thank the Gilman School and their co-teachers and students. Gilman's computer curriculum provides the basis for our work. Special thanks also for those who supported us with their hours of labor and relentless patience -- our families.

For both of us, this manual has special meaning. We dedicate this work to our fathers, Herman Goldberg and the late Warren Culbertson. As products of their enthusiasm and desire to excel, we hope that this book, in some small way, will make them as proud of us as we are of them.

TABLE OF CONTENTS

PAGE

INTRODUCTION Perspectives 1

Section 0.1 The Apple: A Perspective 3
Section 0.2 A Slice Of The Apple: A Perspective 5

CHAPTER ONE Establishing Solid Roots 6

Section 1.1 Control Over A Variable 8
Section 1.2 Strings Attached 13
Section 1.3 Key Assignments 17
At The Keyboard: Cranking It Up 21
Section 1.4 Print Perfect 23
Section 1.5 Mathematical Wizardry 28
Section 1.6 String Things 32
Section 1.7 Tracing The Error-Free Route 35
At The Keyboard: Whoops! 39
A Little Bit More: Details Which Help 41
Review Exercises 43

CHAPTER TWO What If? 44

Section 2.1 Follow Your Nose 46
Section 2.2 Programming Jumps 51
Section 2.3 The First R -- Teaching The Apple To Read 54
At The Keyboard: Disk In Use 58
Section 2.4 Decisions, Decisions, Decisions 61
Section 2.5 We Can Solve All Those Errors, If... 66
Section 2.6 No If's And's Or But's 71
A Little Bit More: Placing It Where You Want It 76
Review Exercises 78

CHAPTER THREE Again And Again And Again 79

Section 3.1 Super Dooper Looper 81
Section 3.2 Another Way Around 85
At The Keyboard: Editing Mistakes Again 90
Section 3.3 Computer Makes Little Things Count 92
Section 3.4 Accumulating Loops 97
Section 3.5 Completely Looped 103
A Little Bit More: Spicing Up The Print Statement 109
Review Exercises 111

PAGE

CHAPTER FOUR Opening New Doors 112

Section 4.1 The Great Chopper 114
Section 4.2 GOZINTA! 118
Section 4.3 Stroll Through Number Theory 121
At The Keyboard: Peeking Into The Future 126
Section 4.4 A Program Per Chance 129
Section 4.5 Gaming Randomly 134
A Little Bit More: Economy Size 139
Review Exercises 141

CHAPTER FIVE New Horizons 142

Section 5.1 Face To Face With Graphics 144
Section 5.2 Doing It Up With Lo-Res 150
At The Keyboard: Paddling Downstream 157
Section 5.3 Magnifying The Possibilities 160
Section 5.4 Moving Into Hi-Res 163
A Little Bit More: The Sound Of Input 169
Review Exercises 171

CHAPTER SIX There And Back Again 172

Section 6.1 An Array Of Possibilities 174
Section 6.2 One Vote For Subscripts 179
At The Keyboard: Principal Files 185
Section 6.3 Sorting It All Out 189
Section 6.4 Go-Go Practice 194
A Little Bit More: More If's 199
Review Exercises 201

INTRODUCTION

PERSPECTIVES

Section 0.1 The Apple: A Perspective
Section 0.2 A Slice Of The Apple: A Perspective

SECTION 0.1 THE APPLE: A PERSPECTIVE

Computers? This is the computer age, but what are they really? How did we come to depend on them? No one would deny that they are of vital importance in current events. Some would say they have intelligence of their own and are taking over the world! Some people give them names and even affectionate nicknames. Is the computer some kind of modern day monster, a human-like machine?

Computers seem so complex that it is easy to give them attributes they don't have. Computers have no reasoning ability or intelligence of their own. They are invaluable tools, but only as a means of processing words, numbers, and information. Without human control, in fact, the computer is like an automobile in the driveway. By itself, the car can do nothing. But it has tremendous potential. What a computer can do, if treated properly, is obey your every command. It can keep track of your finances, act as a burglar alarm, or play a Beethoven symphony. And of course, it can play games. All these and more can be done -- you have CONTROL. To have this control, you must instruct the computer. A Slice of the Apple is directed towards this purpose.

To sample the flavor of the computer's history, we will trace the development and use of the early computers. The evolution of the computer was inspired by man's need to process more and more mathematical operations at a faster rate. Liebniz, a seventeenth century inventor and mathematician, said, "It is unworthy for excellent men to lose hours like slaves in the labor of calculations". All the attempts at producing calculating machines before the twentieth century were tributes to the inventiveness of humans with mechanical processes. The abacus, developed five thousand years ago in China, was the first hand-held and operated calculator. In 1642, Blaise Pascal set up a series of gear-toothed wheels and ratchet levers to create an addition dialing calculator. Thirty years later, Leibniz updated the calculator to perform the four basic operations. Neither machine, however, was dependable or reliable.

The nineteenth century saw immense improvements in the earlier calculator inventions. In 1842, Charles Babbage designed the Difference Engine which included a memory unit, a printout capability, data input, and sequencial programming steps. Working with Ada Lovelace, he attempted to produce the design, but was unable to do so successfully. In 1885, using a macaroni box and meat skewers, Dorr Felt designed and built the first mechanical calculator accurate one hundred percent of the time. Wouldn't you have liked to see that? Five years later, a southern bank clerk, W.S. Burroughs invented an adding machine which printed answers. His machine was to revolutionize the banking and business community. In the same year, an ambitious census taker worked at solving the dilemma of handling data processing in huge quantities. Herman Hollerith developed a Tabulator Machine consisting of a hand-operated card puncher, an electrically sensitive card reader, and a card sorter. Fifteen years later his original one man operation had become the Tabulating Machine Company. Later, under the direction of Thomas Watson, the company changed its name to Internal Business Machines, or IBM.

The advancements in technology in the eighty years of the twentieth century have far surpassed any previous accomplishments. In 1939 Howard Aiken, with some assistance from Watson of IBM, developed the Automatic Sequence Calculator -- MARK I. It was the first general purpose digital calculator. It could solve equation systems and produce mathematical tables. It weighed, however, thirty tons and occupied fifteen hundred square feet of floor space! In 1946, a team of three -- Manchly, Eckert, and Goldstine -- at the University of Pennsylvania, developed ENIAC (Electronic Numerical Integrator and Computer). It contained seventeen hundred vacuum tubes, and could perform five thousand calculations per second. It actually dimmed the lights in the community when operated. ENIAC was the first complex computer and stood as the prototype for the world.

The final step in developing the computer was discovered only months after ENIAC was put into use. John Von Neuman recommended the use of Boolean Algebra logic for computer design. The concept was to employ a binary code, or "0" and "1", to give sequential instruction and store data information. The binary code related to the two computer electrical states of "ON" or "OFF". This concept seems elementary, but it caused rapid progress in the state of the art.

In the 1970's, the most powerful single computer circuit component was produced -- the "chip". The microprocessing chip has the parts of the computer chemically formed on its surfaces. It can be less than a square inch in size and weigh an ounce. The chip drastically changed and widened the applications and accessibility of the computer. It's production led to the manufacture of a line of microprocessor computers. The Apple is one example ot this technology, and here is where our story will take place.

Parallelling the advancement in computer machine technology was the development of computer language. Up to the 1950's all communications with a computer were limited to a special binary code, called Machine Language. It is a language of codes, made up of the numbers one and zero, using Von Neuman strategies. Each character, direction, or storage, was coded a different way, posing serious limitations on problem solving. The letter "A" might be coded 01101100; a B code 01101101.

In 1957, a team headed by John Backus and Irving Ziller developed a programming language based on algebra and a little grammar. The FORTRAN (Formula Translation) language, was developed to allow mathematicians, scientists, and engineers access to computer technology. As part of the computer a COMPILER (or ENCODER) was designed to recode this language into machine language. In 1959, through the auspices of the Pentagon and encouraged by Grace Hooper, a new language was created. COBOL (Common Business Oriented Language) was specifically oriented to handle business applications. In 1964, John Kemeny and Thomas Kurtz at Dartmouth created BASIC (Beginner's All Purpose Symbolic Code). It revolutionized the field by simplifying the structure and syntax of computer language. Since its development, many alterations and dialects have occurred. The essence of simplicity is still there. The programming language of the Apple -- called APPLESOFT -- is one such language dialect in the BASIC language.

There are now a multitude of languages. APL, PASCAL, PILOT, LOGO -- these are relatively new languages each with a different emphasis. Although at first glance each language appears to be different in design and structure, each language must still be converted into machine language. In fact, the computer only understands this language -- a language composed of ONES and TWOS.

SECTION 0.2 A SLICE OF THE APPLE: A PERSPECTIVE

The power to program is the power to change the microprocessor from a machine to a tool. Learning the technical and artistic skills of programming is what this text is about. The chosen language here is BASIC -- specifically APPLESOFT -- but this choice does not limit your programming abilities. For the skills of programming go beyond a specific language.

Our approach is one of problem solving. Each new Section is generated by a programming problem. The solution to this problem requires new terminology, new concepts, and new programming skills. Generally, the Chapters are concerned with programming techniques or with the investigation of special capabilities within Applesoft. Each Section within the Chapters includes problems to be solved using the new information. At the end of each Chapter is a cumulative set of problems employing all the new learned input. We recommend that at least some of these solutions be programmed on the Apple after being formulated on paper. In addition to this work, each Chapter includes two special sections -- AT THE KEYBOARD and A LITTLE BIT MORE. At The Keyboard represents instruction in the actual use of the Apple at the "push this button" level. The placement of these Sections is directly related to hands on needs, not to the specific chapter material. The Little Bit More Sections represent materials not specifically needed for basic programming skills, but are included here because the contents are new tools that add a certain spice and/or elegance in programming.

This text is not the Complete Guide for Programming nor a Complete Tour of the Apple. It is a chance for the beginning programmer to get his/her feet wet. The authors have found that basic programming skills, once established, produce a literate user, motivated to learn and learn.

So go ahead, take the proverbial bite with us and investigate the basics of programming in Applesoft.

CHAPTER 1

ESTABLISHING SOLID ROOTS

Section 1.1 Control Over A Variable
Section 1.2 Strings Attached
Section 1.3 Key Assignments
At The Keyboard: Cranking It Up
Section 1.4 Print Perfect
Section 1.5 Mathematical Wizardry
Section 1.6 String Things
Section 1.7 Tracing The Error-Free Route
At The Keyboard: Whoops!
A Little Bit More: Details Which Help
Review Exercises

SECTION 1.1 CONTROL OVER A VARIABLE

Ultimately a program is a solution. It consists of a sequence of instructions, directions and questions which allow the Apple to perform the required task. Learning the language of these instructions and the techniques for giving them is our purpose.

Our initial programming problem will introduce the most important vocabulary. Our problem is to have the Apple calculate the sum, difference, product and quotient of the numbers 10 and 20. At first glance you might pass this off as a mere hand-calculator exercise. From our solution you will see the complexity of the Applesoft language, and you will get a feel for the processes within a program.

In solving this problem, our first task is to feed the initial values, 10 and 20, into the Apple. Inputting these values is accomplished by use of a variable. A variable, you might recall from your algebra work, is a symbol which represents a given value, and their manipulation is a key capability of any computer. We start by assigning the value 10 to a variable. In Applesoft the variable may be a single letter or an entire set of letters. In the case of a set of letters, only the first three are significant. Select variable names which will assist your understanding of what the variable represents. We use the word LET to introduce this assignment. A possible instruction might then look like:

```
LET X = 10 or
LET VARIABLE = 10 or
LET FIRST X = 10 or
LET FIRST ONE = 10
```

When this direction is executed the Apple inserts the value of the variable in its memory. Notice that the last two possibilities actually assign 10 to the same variable, because the first three letters FIR are significant. Imagine if you will, that this memory appears as a set of mailboxes; each one addressed according to the variable name. Once established this address will have the value 10 stored in it until it is replaced by some other value.

Since we also wish to use the value 20, we can assign this value to another variable. Again the statement might look like:

```
LET Y = 20
```

Note that only one assignment may take place in this LET statement. Each of these must be separate directions to the computer.

Now we are ready to do some calculations. We need to use another set of variables for each of our desired results: sum, difference, product, and quotient. The LET statement can again be used. In a LET statement we can use arithmetic or algebraic expressions. Examples of possible LET statements are:

```
LET Y = X + 3
LET Q = 20 - 10
LET XYZ = (Y - 20) + W
```

In each case, the value of the variable is calculated and then stored in the addressed mailbox. The mailbox labelled Q, for example, has the value of 20 - 10 or 10. Obviously before the value is calculated, the Apple must have the values for any other variables used in the expression. Therefore in the first example, the Apple must have a value stored in address X in order to calculate the value of the expression: X + 3.

What then will our assignment statements be for the problem in question? Remember -- pick variable names to help you identify the value. Also remember each assignment needs to be a separate statement.

In our four statements we have used the variables X and Y stored earlier as 10 and 20 respectively.

```
LET S = X + Y
LET D = Y - X
LET P = X * Y
LET Q = Y / X
```

Notice the unusual symbols for multiplication -- an asterisk, * -- and for division -- the slash bar, / . These strange symbols are simply an accomodation to the Apple keyboard.

Since we can do arithmetic calculations within the LET statement we may have chosen the following four directions:

```
LET SUM = 10 + 20
LET DIFF = 20 - 10
LET PROD = 10 * 20
LET QUOT = 20 / 10
```

If indeed we use these four statements, there is no need to store the values of 10 and 20. Here, it is a matter of preference or need as to whether we store values. At this point the program looks like:

```
LET X = 10
LET Y = 20
LET S = X + Y
LET D = Y - X
LET P = X * Y
LET Q = Y / X
```

We have accomplished the assigning of values and the performing of calculations. Although these values have been calculated by the Apple, we need to have the Apple tell us these values. In this case, we need to have the computer output these values onto our monitor screen. The command in Applesoft which allows for all outputting is PRINT. The PRINT command is the most powerful one in BASIC. What can this PRINT statement do? The PRINT statement can print out a numerical value. It can complete calculations. It can locate variable values. And within the PRINT statement we can control the formatting or the style of the output. Here in our problem, we wish to output the values for the sum, difference, product, and quotient. And where are these values? They are stored in the memory under variable names S, D, P, and Q. What we do is simply, within the PRINT statement, indicate the address of the value we wish to have ouputted. Therefore it may look like this:

```
PRINT S
```

The computer does not print out the letter S, but prints the value of the variable S, 30. The following four PRINT statements will complete our first program:

```
PRINT S
PRINT D
PRINT P
PRINT Q
```

These ten instructions will do the things we require. There is one special addition necessary to complete the job. Within a BASIC program, the instructions are ordered using line numbers. Each statement requires a line number. The computer will complete the instructions according to the sequence of line numbers. Line numbers are important because they allow the programmer to delete, insert, correct, and generally refer to statements in the program. It is customary to number lines by 10's or 5's so that there is ample space to make insertions or changes. With this in mind here is one solution to our problem. (HEY- did you see we slipped the word "programmer" into our discussion- congratulations!)

```
10 LET X = 10
20 LET Y = 20
30 LET S = 10 + 20
40 LET D = 20 - 10
50 LET P = 10 * 20
60 LET Q = 20 / 10
70 PRINT S
80 PRINT D
90 PRINT P
100 PRINT Q
```

If we were to have the Apple execute the program, the output would look like this:

```
30
10
200
2
```

We have accomplished the task, but I bet you are thinking -- "ten lines to get out four values - give me back my calculator!" But hang on, this is only one solution. To every problem there are multiple solutions and even this simple one has lots of variations.

Let's back up and look at the PRINT statement portion. You may be guessing that it may be possible to somehow combine these. That is, perhaps we can print out more than one value at a time. And this is exactly the direction we can go to shorten this program. It is possible to indicate several expressions or items in a print statement and separate them with punctuation. One possibility is with the use of the comma. The statement to replace line 70, 80, 90, 100 would be:

```
70 PRINT S, D, P, Q
```

With this new statement, how is the output altered? An execution of the program yields this as our new output:

```
30              10              200
2
```

You might be squirming in your chair -- but hold on because the reason for this output is simple. A comma is a column-maker. Upon seeing the comma in the print statement -- the Apple formats the output into columns. With the comma there are (you guessed it) three columns. Having filled these, the computer returns to the next line and fills the fourth value -- the quotient -- in the first column of that next line.

Other ways to change the formatting are possible and will be discussed later in depth. Of course the way you want the output to appear will determine the method chosen. Remember you have control over the computer.

With this change our new program appears like this:

```
10 LET X = 10
20 LET Y = 20
30 LET S = X + Y
40 LET D = Y - X
50 LET P = X * Y
60 LET Q = Y / X
70 PRINT S, D, P, Q
```

But we can eliminate even more. The PRINT statement has the capacity to perform calculations prior to outputting a value. The statement PRINT X * Y will yield a single outputted value. The computer searches out the value of X and of Y. It then calculates the product. Only then is that value printed out. Obviously in doing this, the value X * Y is not stored in the memory of the computer. Only through a LET statement can we do this. But if we do not require the four calculations to be stored, our program could be this:

```
10 LET X = 10
20 LET Y = 20
30 PRINT X + Y
40 PRINT Y - X
50 PRINT X * Y
60 PRINT Y / X
```

The output is identical to that of the first program solution, although we know that the storage of values is not the same. A step further and we can combine lines 30-60 using the column-making comma.

```
10 LET X =10
20 LET Y = 20
30 PRINT X + Y, Y - X, X * Y, Y / X
```

Here the output is like the second solution. And we might even shorten this program. If we do not wish to have our two initial values -- 10 and 20 -- stored, our program can be a single statement.

```
10 PRINT 10 + 20, 20 - 10, 10 * 20, 20 / 10
```

The output is the same. The need to store values is determined by the programmer's need. Here this one-lined program does the job for us. We've come a long way in a few short pages. We've learned a tremendous amount about two elementary and essential statement types - LET statements and PRINT statements.

A quick review is in order. The LET statement allows the user to assign a given value to a given storage location. The name or address of the location is a variable. A variable can be a single letter or set of letters (but remember only the first three are used). The value of that variable can be given as a number, a numerical expression, or as an algebraic expression. But the expression is evaluated as a number prior to being stored.

The PRINT statement allows for all outputting. It will output a numerical value. That value can be calculated from a numerical expression or a variable expression. Also the format of the output, with the use of punctuation, can be controlled within the PRINT statement.

Nobody can eat just one slice of Apple. So...ONWARD!

EXERCISES 1.1

1. Write a program which will find the product of 6 and 7 and divide this product by the sum of 6 and 7. Your output should include the product, the sum, and the quotient.

2. Write two different programs to do the following. Find the product of 12 x 12, 12 x 13, 12 x 14, and 12 x 15. The first program should store each product. The second program should find the products without storing any values.

3. A function machine performs one mathematical operation. It has a function rule which acts upon an input and produces an output value. Suppose the rule was "multiply the input by 2 and add 1". If N was the name for the input, then we could rewrite the function rule as: 2 * N + 1. With this function machine, the input value 13 would produce an output of 27. Write a program to simulate a function machine. In your program choose a value for N and calculate the function machine value.

4. Try your hand at creating a function machine. Write a function machine in which you make up the function rule. Choose any input you wish. Be sure to print out both the input and the output values.

5. The following program uses the ideas from our work.

```
10 LET A = 10
20 LET B = A + 1
30 LET C = A * B
40 LET D = C/A
50 LET E = D - A
60 PRINT E
```

A. What is the output of this program?
B. If we replace line 10 with 10 LET A = 15, what value is outputted?
C. Rewrite the program so that the value of A,B,C,D, and E are outputted on five lines.
D. Rewrite the program so that the five values are outputted on as few lines as possible.

SECTION 1.2 STRINGS ATTACHED

With our new vocabulary -- LET and PRINT-- we would like to approach a new programming problem. In this new program we wish to further explore the idea of a variable.

Just when you think you've pleased your math teacher, your English teacher demands something more. She demands a "transmogrification" (look THAT up in your Webster's!) of our calculator program. She wants words. Modify our program from section one so that the program calculates the sum and product of 10 and 20 and then outputs the results in the following manner:

```
SUM OF 10 AND 20 IS 30
PRODUCT OF 10 AND 20 IS 200
```

Although we already have a program to do the calculations, we have the additional obstacle of outputting "words". In computerese the term for a collection of letters, symbols, words, or sentences is STRING. In our problem, the words SUM OF is a string; the word AND is a string. To the Apple a string is handled as if it is an expression. To signify that it is a string expression, quotations are placed around the string. Anything inside the quotation marks is considered as an element in the string- that includes spaces, symbols, or punctuation. Several short programs will help illustrate strings.

```
10 PRINT "WHAT'S TRANSMOGRIFICATION?"
```

Here is a PRINT statement using a string. When executed the output is the following:

```
WHAT'S TRANSMOGRIFICATION?
```

Although the Apple does not have any conception of what the string means, it is printed out as any expression would be. Note that it is ouputted letter for letter, space for space. The quotation marks are not printed, however. They indicate a string, but are not actually part of it.

Take a look at the next short program and make a guess as to how the output would appear.

```
10 PRINT "WHAT'S" , "TRANSMOGRIFICATION?"
```

Here we have two strings separated by that column-making comma. The output is:

```
WHAT'S          TRANSMOGRIFICATION?
```

The strings are outputted exactly as they appear, but the strings are placed in the first and second columns.

In the next short program the two strings are placed together without benefit of punctuation.

```
10 PRINT "WHAT'S" "TRANSMOGRIFICATION?"
```

The output looks like:

The two strings are put in adjacency. Note that because a space was not part of either string, no space was placed between the strings in the output.

Strings can be placed along with algebraic/numerical expressions within the PRINT statement.

```
10 PRINT 5 "TH GRADERS ARE ON VACATION"
```

The 5 is printed out followed by the string. When executed, the printout is:

```
5TH GRADERS ARE ON VACATION
```

Notice that the value 5 and the beginning of the string -- the letter T -- are placed next to one another. In order to get a space, two alternatives can be used. First, an additional space may begin the string, instead of the T.

```
10 PRINT 5 " TH GRADERS ARE ON VACATION"
```

As a second option, an entirely new string consisting of a single space may be inserted between 5 and the string.

```
10 PRINT 5 " " "TH GRADERS ARE ON VACATION"
```

The output would consist of a 5, followed by the string space, followed by the other string.

We are prepared to finish our solution to the problem. This program prints out the two calculated values.

```
10 LET X = 10
20 LET Y = 20
30 PRINT X + Y
40 PRINT X * Y
```

To each we can add strings to our PRINT statements. For example line 30 now becomes:

```
30 PRINT "THE SUM OF "X" AND "Y" IS " X + Y
```

When executed, this output would have first the string THE SUM OF (note the space at the end of the string). Then the value stored in X is printed -- 10. Remember X is a variable, not a string. In order to print out the letter X, it would have to be in quotations too. Printed next would be the AND with spaces before and after. Then printed is 20 (the value of Y); then the string IS (again with spaces); then the value of the algebraic expression X + Y -- 10. The final program would look like this:

```
10 LET X = 10
20 LET Y = 20
30 PRINT "SUM OF "X" AND "Y" IS " X + Y
40 PRINT "PRODUCT OF "X" AND "Y" IS " X * Y
```

You might be asking yourself -- "Since strings are handled like an expression in PRINT statements, can a string be stored in a mailbox?" The answer is simple -- YES. But the set of mailboxes for storing strings is separate from numerical variable values. Therefore we must use a different notation for indicating its address. The variable which tells the address for a stored string is called a STRING VARIABLE. (In computerese -- ALPHANUMERIC VARIABLE.) It is designated by a letter or set of letters followed by a dollar sign -- A$ (read A-STRING), FIR$, AMERI$. Again, as with a numerical variable, only the first three letters are significant to the computer. To store a string we can use the LET statement. An example:

```
10 LET A$ = "WHAT'S TRANSMOGRIFICATION?"
```

The string is still placed in quotations. The address would be A$. The string could be a word, letter, sentence, punctuation mark, or even a space. LET SP$ = " " is a statement for storing a space string.

A string is worth storing if it is repeated within a program. For example in our program, the string "AND" and the string "IS" are used in the two PRINT statements. To save time, both of these strings could be assigned string variable names.

```
25 LET A$ = " AND "
28 LET B$ = " IS "
```

Then within the PRINT statements, if a string variable appears, the Apple outputs the value of that string. The following PRINT statement will yield the string " AND ":

```
30 PRINT A$
```

With line 25 and 28 inserted in the program, our new line 30 might look like this:

```
30 PRINT " THE SUM OF ";X;A$;Y;B$;X + Y
```

You see something unique in this statement -- the semicolon. The semicolon is another formatting punctuation mark, much like the comma. The semicolon used within a PRINT statement simply packs one expression next to another. That is, the value for X is placed adjacent to the string A$. You might remember that this can also occur when one item of the PRINT statement is placed next to another without any punctuation. But here, the semicolon is needed. Whenever two variables, whether string or numerical are placed next to each other in a PRINT statement, a semicolon must separate the items. This is to avoid confusion as to the variable name. For example, 10 PRINT BA$, is read as one string variable, as opposed to 10 PRINT B; A$. To help with any confusion, it might be wise to separate all items in the PRINT statement with semicolons.

Completing the program might look like this:

```
10 LET X = 10
20 LET Y = 20
25 LET A$ = " AND "
28 LET B$ = " IS "
30 PRINT "THE SUM OF ";X;A$;Y;B$;X + Y
40 PRINT "THE PRODUCT OF ";X;A$;Y;B$;X * Y
```

The execution of the program yields this:

```
THE SUM OF 10 AND 20 IS 30
THE PRODUCT OF 10 AND 20 IS 200
```

With the completion of our second program solution, let's review the notion of a string. A string is an expression made up of characters -- letters, symbols, punctuation, or spaces. It is indicated by placing it within quotations (It can be, by the way, 256 characters long). The string expression can be outputted using the PRINT statement. It can be stored using a string variable name in a LET statement. The Apple's ability to handle strings is one feature which certainly distinguishes it from a hand held calculator.

EXERCISES 1.2

1. Write a program to create the headlines of a scandal sheet newspaper -- the National Intruder. Store the names of two of our friends. Without storing any other strings, create a program with the following output:

```
AN INTRUDER SCOOP
______ AND ______ WERE DISCOVERED TOGETHER.
______ DENIED RUMORS OF A ROMANTIC AFFAIR.
______ IN JEOPARDY OF LOSING HIS JOB.
DETAILS INSIDE
```

2. The corner cafe is being computerized. Write a program to assist the management. Store the five entrees and their prices. Then print out a menu.

Example: QUICHE LORRAINE ... $3.25

4. Write a program to store your first, middle, and last names in three separate addresses. Then output them in bibliographical form: LAST NAME, FIRST NAME, MIDDLE NAME

5. Create the first alliteration writer. An alliteration is a phrase or sentence in which each word begins with the same sound.

SECRETARIES SELDOM SELECT SKIMPY SALADS.

In a program store five words starting with the same sound. Then print out four different sentences, each using the five words in a different order.

6. You are being put to the test. List the exact output produced by each PRINT statement in this program.

```
10 LET A$ = "CONFUSION"
20 LET B$ = "STRIKES"
30 LET C = 6 + 4
40 LET Q = .5
50 LET Z = C * Q
60 PRINT C,Q,Z
70 PRINT A$;B$;C;" TIMES "
80 PRINT Z;" ";A$
90 PRINT Z,C,B$
100 PRINT "WE HOPE WE HAVEN'T CAUSED ";A$
```

SECTION 1.3 KEY ASSIGNMENTS

We have learned to assign values to variables using the LET statement. Now we can explore an alternative way to assign values.

Blaise, age four, has built his first robot -- FUNCTION ONE. It is actually a single task machine. It can only perform one skill. It allows an adult (and adults ONLY) to input a number by pushing keys on a calculator glued to the robot's chest. FUNCTION ONE goes through some gyrations before outputting -- across his eyes -- the output value. In mathematics, we call this input-output process a function machine. Write a program to allow the keyboard user to input a value into a function machine -- F(X) = 2X + 1. Print out the function machine's input and output.

Unlike our previous programs, this program allows for a change in the assigning at the keyboard. In Section 1.1 we wrote a program to do this task, but assigned the input value within the program.

```
10 LET X = 6
20 LET F = 2 * X + 1
30 PRINT F
```

In order to change the input to the function machine, the programmer would need to replace line 10. Our new technique will not assign a specific value within the program. It will allow the person at the keyboard to make the assignment during the execution of the program. The necessary statement type for doing this is:

```
10 INPUT X
```

The INPUT statement is another form of the assignment statement, and therefore part of the statement must be the variable name. This is the address in the mailbox where the value will be stored. Upon being executed, the Apple will pause and wait for the operator to punch in the value of the variable. On the screen a question mark and a flashing rectangular box appear. The execution of the program will not proceed until an acceptable value is typed in. Upon typing in a number (and pressing the return key) the execution of the program will proceed.

With this new statement 10, here is our program:

```
10 INPUT X
20 LET F = 2 * X + 1
30 PRINT F
```

Here is a the program being executed:

```
? (Here the Apple was waiting for input for X).
```

Then after 10 was typed in, the program finishes.

```
21
```

Several additional things can be said about the INPUT statement. First, it is possible to use string variables as well as numerical variables. The statement would look like: 100 INPUT A$. Obviously it is important that the person at the keyboard be cognizant of which kind of value should be typed in.

Secondly, it is possible to assign more than one value within a single INPUT statement. You will remember that this cannot be done with a LET statement. But in the INPUT statement several addresses may be named. The variables in the INPUT statement need to be separated by commas. And the Apple will indicate the need for two values by repeating the question mark response. Indeed the program will not proceed until all values have been assigned. Even a combination of string and numerical variables is possible: 200 INPUT A, B, C$

At this point, prior to completing our program solution, we would like to emphasize the need for clarity in all output. This need takes two forms. First and foremost, all PRINT statements should be created so that the final output is readable and understandable to the operator. To our way of thinking, a single number -- 21 -- on the screen as a final output is unacceptable. Be creative with how the output is formatted. Style is important here. Several examples of possible outcomes for our function machine program are explained below.

Example 1

```
FUNCTION MACHINE
F(X) = 2 * X + 1
INPUT           OUTPUT
10              21
```

This example is called the column or label form of output. The initial output comes from PRINT statements containing the necessary strings. Placement may come from spacing. The INPUT-OUTPUT came from placing these strings in a PRINT statement with commas. The numbers are printed, likewise, in columns.

```
30 PRINT "FUNCTION MACHINE"
32 PRINT "F(X) = 2 * X + 1 "
34 PRINT "INPUT", "OUTPUT"
36 PRINT X, F
```

Example 2

```
WHEN 10 IS INPUTTED IN THE FUNCTION, THE OUTPUT IS 20.
```

This is an example of sentence or phrase form of output. The output is created by PRINT statements which combine strings and variables. The only hazard is the length of a PRINT statement. Since strings have no meaning, it is possible that the string would be arbitrarily divided, perhaps in the middle of a word. The monitor line is 40 characters long. You might divide the sentence where you want it to be divided and make a new PRINT statement for the additional part. Hence the print statements here may be more productive as two:

```
30 PRINT "WHEN ";X;" IS INPUTTED IN THE FUNCTION,
         THE OUTPUT IS ";F;"."
```

```
30 PRINT "WHEN ";X;"IS INPUTTED IN THE"
32 PRINT "FUNCTION, THE OUTPUT IS ";F;"."
```

Output is so crucial that it must never be dismissed without contemplating all possibilities. Here is an opportunity to be creative.

The second area of clarity in output is within the INPUT statement. Each time an INPUT statement is encountered, the keyboard user is required to place in a value. Without a complete understanding of what is taking place, that operator may not know what kind of value is wanted or even acceptable. He might be asking himself: "Do I input a string?"; "What is the range of values that are acceptable?" It is imperative that as part of the INPUT statement clear directions are given. Those directions can take two forms. First you can give directions about input by using a preceding PRINT statement. The operator has output on the screen (from the PRINT statement execution) and then receives the flashing box and question mark demanding input value. It might look like this:

```
5 PRINT "TYPE IN A NUMBER FROM 1 TO 100"
10 INPUT X
```

At the time the user must type in a value, it looks like this on the screen:

```
TYPE IN A NUMBER FROM 1 TO 100
?█
```

Another example of this:

```
5 PRINT "TYPE IN YOUR NAME (FIRST NAME, SECOND NAME)
10 INPUT FIRST$, LAST$
```

```
TYPE IN YOUR NAME (FIRST NAME, SECOND NAME)
?█
??██
```

The alternative to the preceding PRINT statement is to use a capability of the INPUT statement. It is acceptable in an INPUT statement to precede the variable names with an item in quotations. This item acts the same as a string in a PRINT statement. That string is printed out, followed only by the flashing rectangle. A semicolon must be used to separate the string item from the variable name. Be careful not to try to include as part of the printed item a variable -- for this variable will act as the requested input address. You might also include a space or some form of punctuation as part of the string.

```
10 INPUT "TYPE IN A NUMBER FROM 1 TO 100 - ";X
```

It looks this way on the screen. Note the placement of the rectangle on the same line as the printed out string. This is a special feature of the INPUT statement designed to allow for clarity.

```
TYPE IN A NUMBER FROM 1 TO 100 - █
```

Another example:

```
10 INPUT "WHAT IS YOUR FIRST NAME, LAST NAME: " ; F$, LS
```

```
WHAT IS YOUR FIRST NAME, LAST NAME: █
```

We have thrown many ideas at you at once. Input is a command to allow the user to assign any value, string or numberical, to a given variable. This can be a string or numerical value. We have attempted to stress the importance of clear, understandable and stylish output using the PRINT statement capabilities and the special print capability of the INPUT statement.

Here is one possible programming solution then:

```
10 INPUT "TYPE IN THE INPUT FOR FUNCTION MACHINE: ";X
20 LET F = 2 * X +1"
30 PRINT "FUNCTION -- F(X) = 2X + 1"
```

```
40 PRINT "INPUT:";X , "OUTPUT:"; F
```

When executed it might appear like this. This time the operator inputted the value 19 for X.

```
TYPE IN THE INPUT FOR FUNCTION MACHINE: 19
FUNCTION -- F(X) = 2X + 1
INPUT:19        OUTPUT:39
```

You should feel so accomplished -- you've outdone Blaise (of course he's only four years old!).

EXERCISES 1.3

1. Write a program to simulate FUNCTION ONE. Make up your own function rule. Get the input from the keyboard user. The input and output should be printed as an ordered pair.
Example: (4,9)

2. The corner fruitstand -- APPLAND -- uses a computer program to calculate sales totals. Ask the customer for the number of each of the four types of fruits sold -- cherries, pears, peaches, and apples. Then calculate the total cost.

3. Our generation still has trouble using the metric system to measure. We still need to convert each metric measurement into the English system equivalent. Write a program to input a measure in meters and convert it to inches.

4. To show off the Superhuman adding ability of Captain Apple, write a program to allow the keyboard operator to input ten different numbers. Then stand back and watch the amazed look as Captain Apple computes the sum as fast as he/she can hit the RETURN key. After printing out the sum, also have the Captain say something profound, like: "GEEZY PEEZY, THAT WAS NOTHING!"

5. MADLIBS are incomplete sentences in which the player, without seeing the sentence, supplies the missing words. Write a program to ask the player for several missing words. Within the INPUT statements, specifically request the part of speech needed for the blank -- i.e. noun, verb, adverb, etc. After the words have been supplied print out the MADLIB sentence. For example:

```
A PINK ______ MOVED ______ ACROSS
       (noun)       (adverb)
THE ______ SKY.
 (adjective)
```

AT THE KEYBOARD SECTION 1A

CRANKING IT UP

After our first sections, you've had a chance to write some programs. You are a computer programmer. But, you are saying to yourself..."LET ME AT IT!" You can't wait to have your fingers tickle the keys, and to see your programming skills flash onto the monitor screen.

You've come to the right place. The AT THE KEYBOARD sections are designed to help with your actual hands-on contact with the Apple. We will be exposing you to the "how to's" for controlling the Apple itself, the keyboard, and the disk drives. These Sections act as a reference for your computer time.

You're sitting at the Apple for the first time. You have a written program in hand, but what now? Several things to do first. Familiarize yourself with directions for "POWERING UP", turning on the Apple. Next, familiarize yourself with the keyboard. Although it is similar to a typewriter keyboard, there are several important features to point out. There are really three sets of characters. Simply pressing character keys (similar to the lower case set on a typewriter) will print upper case letters. A second set of characters can be generated by holding the shift key down while pressing the other character keys. The "$" is an example of a shift character. The third set is generated by holding down the CONTROL (CRTL) key and pressing another character. These are referred to as control characters and we will be using them throughout our work. There are also several unusual keys -- ESC, RESET, ←, →, which will be introduced as needed. Finally, familiarize yourself with the CURSOR. When turned on, a small flashing rectangle will appear at the bottom of the screen. This is called the cursor and is an indicator of where typed characters will appear. The small bracket] at the beginning of the line is an indicator of the language which is to be used -- in this case Applesoft.

Prior to typing in the program, we must familiarize you with one other topic. There is a set of commands, not part of the program, which instructs the Apple in how to handle the program. These are called SYSTEM COMMANDS. Without them the computer cannot interpret your typed program. The first of these needs to proceed each new program. Type in:

```
NEW
```

The NEW command clears the memory banks of all previous statements and all variable values. It should be the first statement typed to avoid any statements that were part of an old program becoming part of the new one.

With that done, you can begin typing in your program. Carefully type each line, with line numbers. When you finish a line, press the RETURN key to return the cursor to the start of the next line. Only when the RETURN key is pressed will the statement be entered into the Apple. It is as if you are giving the "OK" for the line to be included in the program.

At any part in your program typing, you may use another SYSTEM command -- LIST. The LIST command will print whatever statements are currently in the computer memory. It will allow you to see what you have typed. It will be a helpful tool in making program corrections. It is wise to LIST the entire program once finished to proof your program.

The most crucial SYSTEM COMMAND is used once the program is typed in. The RUN command instructs the Apple to execute the statements in the program. It is the ON button for the program. Whatever the result, RUN will complete your instructions step-by-step. We often say that a program is "run" or that you should "run" your program.

The following is an example of what you might see the first time you have your hands on the Apple. Our first time programmer typed in his program, listed it as a check, and then executed it.

```
]NEW

]10LETA$="APPLE"

]20LETB$="TRIAL RUN"

]30PRINTA$,B$

]40PRINT"HOW ABOUT THAT?"

]LIST

10  LET A$ = "APPLE"
20  LET B$ = "TRIAL RUN"
30  PRINT A$,B$
40  PRINT "HOW ABOUT THAT?"

]RUN
APPLE           TRIAL RUN
HOW ABOUT THAT?
```

Note several things. When the LIST command is typed, the Apple formats the statement nicely with spaces in appropriate locations. This will keep you from typing needless spaces. Remember, spaces are necessary only when they are part of a STRING. Also after each line typed, the RETURN key was pressed, but it does not have an observable effect on the screen. Here is a chance to see the size of the columns, caused by our column maker comma.

There's nothing more to say except -- POWER UP -- and GET TO IT!

Not to interrupt your hands-on experience, but a few comments about programming modes. When we speak of programming, we are simply referring to a list of instructions. It is possible to perform a single instruction at a time. For instance, if we type PRINT "HELLO", the word HELLO will immediately be printed out below the typed lines. If we type the statement LET X = 5, without a statement number, the value X will be stored in mailbox X. It is just as if your mother asked you to empty the trash can or put away the dishes. She has given you an instruction. This form is called the IMMEDIATE mode. The computer, upon seeing this, will immediately execute the instruction, just as you would carry out mother's instructions immediately (well...maybe!).

The form we have been discussing in our work is the PROGRAM mode. The analogy of this form is an entire list of instructions that you may find on the kitchen table upon returning home from school: clean your room, empty the trash, do your homework, etc. If a computer was reading the list of jobs, it would carry out each one in order quickly and efficiently. It would do them in order of the line numbers that correspond to each job. And it would only do the jobs after we give the OK -- by typing RUN.

Most programming task require a complete set of instructions to solve the problem. We can reserve the use of the immediate mode for simple calculations.

```
PRINT "UPWARDS AND ONWARDS"
UPWARDS AND ONWARDS
```

SECTION 1.4 PRINT PERFECT

With the emphasis on creative and descriptive output, we are going to focus our attention on formatting. We know that formatting is controlled within the PRINT statement. And we already know a little about how commas and semicolons between items in the PRINT statement affect output. Our new programming problem will help us in discovering more.

Write a program which will play a modified simple game of NAME THAT WORD. You will hide the letters of a word in various assignment addresses and then print out those letters one at a time. We'll see which classmate can guess the word first. Obviously we want those letters on the same line. (A title would improve the look of the program.)

This program itself should not cause much difficulty. We are storing letters using string variables, and then outputting them through PRINT statements. The statements below give a sample for the word Apple.

```
10 LET A$ = "A"
20 PRINT A$
30 LET B$ = "p"
40 PRINT B$
50 LET C$ = "p"
60 PRINT C$
70 LET D$ = "L"
80 PRINT D$
90 LET E$ = "E"
100 PRINT E$
```

However, this really doesn't satisfy our requirement. Although the letters are printed one at a time, the output is vertical in nature.

```
A
P
P
L
E
```

So let's explore some possible ways of correcting the flaw. Your first guess might be to combine the PRINT statements into one.

```
100 PRINT A$; B$; C$; D$; E$
```

But this solution is avoiding the issue, for the letters are being printed out together. The output -- APPLE -- is not satisfactory to our game. We want separate PRINT statements, but with each letter of the output on the same line. Under normal circumstances, each time the Apple sees a new PRINT statement it skips to the beginning of the next line. That is what produces the vertical output of the first draft. It is possible to have it ignore this impulse, and the technique involves our previously learned punctuation. If at the end of the PRINT statement items there is a trailing comma or semicolon, the Apple will hold the output on the same line when it sees the next PRINT statement.

```
20 PRINT A$;
40 PRINT B$
```

Here in this example, after outputting the A (stored in A$), the computer would remain on the same line. The items printed in the next PRINT statements would simply follow that A. The end result would be -- AP. In this example, since there is no trailing punctuation in line 40, the next PRINT output would be at the start of the next line.

A trailing comma has the same effect. It simply says to place the next print items in the next column over.

```
20 PRINT A$,
40 PRINT B$
```

This would result in the output:

```
A          P
```

Remember that there are three columns to be filled.

So we are ready to complete the first part of the program by placing trailing semicolons at the end of each PRINT statement.

```
10 LET A$ = "A"
20 PRINT A$;
30 LET B$ = "P"
40 PRINT B$;
50 LET C$ = "P"
60 PRINT C$;
70 LET D$ = "P"
80 PRINT D$;
90 LET E$ = "E"
100 PRINT E$
```

You may be asking yourself if there are other ways in which a programmer can control the output from within the PRINT statement. Although the semicolon and comma are the only punctuation controls, there are two important special functions or capabilities that can be employed. These functions relate directly to our output of a title. The two functions are TAB and SPC.

The TAB function works similarly to the TAB button on a typewriter. With the TAB function the programmer designates the horizontal position at which the output should begin. We would write TAB (8) -- read TAB-8. The "8" inside the parenthesis is called the argument of the function. In our function machine notation this argument would be the input. This argument may be a numerical or algebraic expression. Within the PRINT statement this will cause the output to begin at the 8th position horizontally. The statement might look like this:

```
10 PRINT TAB (8); "BASIC"
```

Note the use of the semicolon to separate the items. The output is the word BASIC, but printed out starting in the 8th column.

Compare the output of these two statements.

```
5 PRINT "BASIC"
10 PRINT TAB (8); "BASIC"
```

```
BASIC

        BASIC
```

You can see how the starting position is affected. The programmer can use the TAB function to set up his own column formatting by setting the starting location of the titles.

```
100 PRINT TAB (5); "INPUT"; TAB (25) "OUTPUT"
```

The execution of this statement results in this:

```
INPUT                    OUTPUT
```

Several things need to be kept in mind. Remember that there are 40 columns across the page. An argument greater than 40 will simply be continued onto preceding lines. That is, TAB (45) will result in an entire line skipped and then will have output starting at the 5th column of the next line. Also the programmer should be careful not to try to force the output on a position that has already been passed. The computer can move only left to right; it cannot backspace to a smaller horizontal position on a line.

```
10 PRINT TAB (5); "APPLE"; TAB (7); "SOFT"
```

Here the 7th position has already been passed once APPLE is printed out. With such a statement, the TAB(7) is simply ignored.

One can see how the TAB function could help in printing titles. To center a title, one can select a TAB input to move it to the desired location. To center NAME THAT WORD on the screen output, we want to set our tab at half the difference of 40 minus the number of letters.

```
PRINT TAB (13); "NAME THAT WORD"

             NAME THAT WORD
```

The other function that can be used in controlling output is SPC (the initials for space). This function operates like TAB. It needs an argument. This time the argument indicates the number of spaces between items.

```
10 PRINT SPC (8); "BASIC"
```

Here the Apple will skip 8 spaces prior to outputting BASIC.

```
20 PRINT "APPLE"; SPC (10); "COMPUTER"
```

In this example there will be 10 spaces between the APPLE and COMPUTER. This function is a substitute for a string of spaces in quotes discussed earlier.

How does this program differ from 20 PRINT "APPLE"; TAB(10); "COMPUTER"? The difference is obvious when the output of both programs is examined.

```
TAB:  APPLE____COMPUTER

SPC:  APPLE__________COMPUTER
```

Notice that the TAB function always starts counting from the left margin. The SPC command, however, starts counting from the last letter of the previous word. If we add these commands to our program, things are really starting to get fancy!

Here the words "NAME THAT WORD" have been centered over the other output.

```
10 PRINT TAB(2);"NAME THAT WORD"
20 LET A$ = "A"
30 PRINT A$; SPC (3);
40 LET B$ = "P"
50 PRINT B$; SPC (3);
60 LET C$ = "P"
70 PRINT C$; SPC (3);
80 LET D$ = "L"
90 PRINT D$; SPC(3);
100 LET E$ = "E"
110 PRINT E$
```

```
 NAME THAT WORD
A   P   P   L   E
```

With almost total control, it is fun to imagine all the things we will be able to do. And now, you begin to sense how we can get the Apple to sit up and beg-- at our command.

But before leaving formatting, one more little bit of styling. In our program above, we have the title jammed up against the output. Would it not be nice to have the power to skip lines in the output? And what's more, wouldn't it be nice if we already had that tool? Yes indeed, we have that tool in our PRINT statement. Each time a new print is executed, the Apple skips to a new line. If we introduce a PRINT statement without any items we are forcing the Apple to skip to a new line. Then the next PRINT statement once again will skip to a new line before outputting. The end result -- a skipped line. We affectionately call this a dummy PRINT statement.

```
15 PRINT
```

If we insert line 15 into our program, you can see how it affects the output:

```
 NAME THAT WORD

A   P   P   L   E
```

We now control horizontal and vertical positioning. We control the line on which we are printing through the use of trailing punctuation and dummy PRINT statements. And we control output positioning with the TAB and SPC functions. We approach our new tasks to come with the tools to make our output creative, stylish, and well-defined.

EXERCISES 1.4

1. What is the exact output of each program below:

A.)
```
10 PRINT 2 * 4, 2 + 3
20 PRINT 8-4, 6 + 2, 15/5
```

```
B.)      10 PRINT "HELLO"
         20 PRINT
         30 PRINT TAB(5); "GOODBYE"

C.)      10 PRINT "SKIP","TO",
         20 PRINT
         30 PRINT "NEXT LINE" ;
         40 PRINT SPC(20); HERE
```

2. The following incomplete program allows the user to input a TOP TEN ROCK SONG and the artist(s) who sings it. Use SPC, TAB and/or dummy PRINTS to output the information in column form with appropriate headings. Then redo the PRINT statements so that the information is outputted in sentence form.

```
10 INPUT "INPUT A TOP TEN ROCK SONG :";A$
20 INPUT "INPUT THE ARTIST :";B$
```

3. The following values have been stored in a program:

A=4, B=9, C$="*", D$="="

Write a PRINT statement to produce each of the following outputs.

```
A.) 4 * 9 = 36
B.) THE PRODUCT OF 4 AND 9 IS 36
C.) 4 TIMES 9 = 36
D.) 4 * 9 = 9 * 4
```

4. Write a program which gives instructions to a keyboard user to input any number. The program should increase the input by 5 and double that result. Be certain to explain to the user what the program has done.

5. Write a program that will produce the following output.

```
A
 P
  P
   L
    E
IS GREAT FUN!
```

SECTION 1.5 MATHEMATICAL WIZARDRY

The next two sections are a look at some of the special functions of the Apple. In this one, we will investigate the mathematically oriented functions.

Mathematics teachers are always easily impressed by eager students. Be the first to write a program to input a number and print out its absolute value, square and square root.

A little inventory is called for here. Already we have discussed four mathematical capabilities of the Apple. Those are the four basic operations -- addition, subtraction, multiplication, and division. Remember the symbols for each: +, -, *, /. We would expect these, but what of others?

We would like to begin our discussion with the three that are part of the program problem: powers, square roots, and absolute values. The Apple has the ability to raise an expression to a power. Without the ability to type exponents, the notation is slightly different. An upward arrowhead stands for "is raised to the power of". For example, 3 ^ 2 means three raised to the second power (or squared); A ^ 5 means the value of A raised to the fifth power. The power may be a numerical or algebraic expression as well, but it will need to be in parentheses: X ^ (2 * 3 + 1)

The computer is also able to compute square roots. In case you've forgotten, the square root of a number, b, is the number a, such that $a^2 = b$. So, 3 is the square root of 9 ($3^2 = 9$). The square root function is handled similarly to the TAB and SPC functions. The functional symbol is SQR; the argument in this case is the number for which you want the square root. SQR (9) means the square root of 9 ($\sqrt{9}$); in this case 3. Again, the argument may be an expression. That expression is evaluated prior to carrying out the square root function.

Absolute value is defined as the positive distance a number is from 0. Both 5 and -5 have absolute values of 5. Generally, $|X| = X$ if X is positive or zero; $|X| = -X$ if X is negative. The Apple is capable of finding the absolute value of an expression. The notation is: ABS (-5). The argument is the value to be used and may be any algebraic or numerical expression. The value of ABS (-5) is 5.

With this special function now available to us, we are prepared to write our solution. We may wish to title the columns -- ABS, SQR, SQUARE. This has been done in line 20 of the program below. We want to allow the user to input the value used. Using our TAB function to establish the columns is done within the PRINT statements.

```
10 INPUT "NUMBER: "; N
20 PRINT "NUMBER"; TAB (10); "ABS VALUE"; TAB (20);
         "SQUARE ROOT"; TAB (30); "SQUARE"
30 PRINT
40 PRINT N; TAB (10); ABS (N); TAB (20); SQR (N);
         TAB (30); N ^ 2
```

The operator keyed in 25, and the output looks like:

```
NUMBER: 25
NUMBER    ABS VALUE SQ ROOT   SQUARE

25        25        5.        625
```

Did you earn "brownie points" with your math teacher?

Before leaving a discussion of mathematical functions it is essential to consider the order in which operation/functions are carried out. For example, what value would the following expression have?

20 / 4 * 2 ^ 2

If your answer is 100 or if you are uncertain, you will be surprised to find the result is 20. The reason for the answer lies in what mathematicians call the order of operations. It is not true that we simply go from left to right performing the indicated operations. Instead operations and functions have been given a special priority. The priority for evaluating is:

A.) Special functions -- these include ABS, SQR, and powers.(With more sophisticated mathematics, you'll learn others.)
B.) Multiplication or division
C.) Addition or subtraction

When performing several functions of equal priority, they are carried out from left to right.

In our example, first 2 was squared (yielding 4). Then the division 20 / 4 was performed (yielding 5). Then the multiplication of 5 * 4 yielded a value of 20.

Although this order of operations is easy to learn, there is one other exceptional rule. There is a mathematical means of changing the priority of operations -- grouping symbols. Unlike the many grouping symbols in mathematics-- parentheses, brackets, braces, bars, etc., there is only one symbol used by the Apple -- parentheses. The parentheses tell us to ignore all or part of the order of operations. Instead they indicate that we should perform the operations within parentheses first. If you like, consider parentheses as the overruler.

With that in mind, try to evaluate this:

(20 / 4 * 2) ^ 2

Although we would normally square first, here we are to first perform the operations 20 / 4 * 2. Within the parentheses the order of operation prevails. Here with division and multiplication if means going from left to right -- the value of 10. Once all operations have been performed within the parentheses then the others are performed, still according to the order of operations. The value here is 100. Also, there is a priority of parentheses themselves. The operations within the inner-most parentheses are performed first. If the sets of parentheses are not within each other, then left to right preference is honored.

With all the rules and their exceptions, it will be worth your time to evaluate these expressions. Be careful to use the correct order of operations. Do not try to do all the calculations in one step. Don't look at the answers until you have completed them.

A.) 12 / 2 * 3 + 2
B.) 12 / (2 + 4)
C.) 9 ^ 1 / 2
D.) 2 + 4 * 3 / 2 * 5 - 3
E.) 3 + (4 - 2) * 6 + 4 / 2
F.) 2 (1 + 3 (7 + 4 * 2))

(ANSWERS: 20, 2, 4.5, 29, 17, 92)

If you are still faltering, seek help with your math text. You'll be happy to know that the Apple follows these exact procedures in evaluating expressions. (And it does so without a mistake.) But it is important to be able to predict (or at least estimate) reasonable values.

Speaking of values, sometimes the Apple returns expression values which are surprising, even confusing.

```
10 PRINT 8 + 16 / 48

.5
```

In this example, you might have expected the fraction $\frac{1}{2}$ or 1/2. It is essential to understand that the Apple performs all computations using decimals, and all output will be in decimal form. Hence the output above is .5 instead of 1/2. This does lead to problems. It is possible that the output may be a repeating decimal: .383838... Without the capability to handle this, the computer does the next best thing -- round off. The Apple can handle a number with nine digits. Those nine may be after the decimal point, before the decimal point, or some combination. After those nine places, the Apple rounds off:

.3838... becomes .383838384
401.876666... becomes 401.876667

When the number becomes very small or large, the Apple approaches them using scientific notation. Scientific notation is used in the output if the number is less than 10^{-2} or greater than 10^{9}. The notation looks quite unusual but is quite easily understood.

.000276 becomes 2.76E-04
12,760,000,000 becomes 1.276E+10

As in all scientific notation one factor is expressed as a number between 1 and 10. The other factor is some power of 10. The "E" stands for "10 raised to the power of". The number which follows is the negative or positive power of 10.

2.76E-04 means 2.76 * 10^{-4}
1.276E+10 means 1.276 * 10^{10}

Now you'll be familiar with this when it appears as part of your output.

You might be asking yourself if there is a limit to the size of a number. The Apple cannot handle numbers greater than or equal to 2E+38. (That's two hundred undecillion -- 2 followed by 38 zeroes!). A programmer has lots of room to play with numbers!

Numbers are among the computer's favorite things to juggle. The Apple handles mathematics quickly and flawlessly. The Apple is equipped with several special mathematical functions: ABS, ^, SQR as well as the four basic operations. It completes multiple operations using the same order of operations we would follow. The output appears always in decimal form, although the output may be rounded off or expressed in scientific notation.

EXERCISES 1.5

1. Predict the output of the following program. Remember that the Apple follows the ORDER OF OPERATION.

```
10 PRINT "3 + 5 * 2 = " 3 + 5 * 2
20 PRINT "(2 + 3) * 4 = "(2 + 3) * 4
30 PRINT "2 * 13 - 4 * 2 = " 2 * 13 - 4 * 2
40 PRINT "3 * 2 ^ 4 = " 3 * 2 ^ 4
50 PRINT "7 + 8/2 - 3 * 2 - " 7 + 8/2 - 3 * 2
```

2. The following values were the output of a program. Convert each number to its equivalent decimal form.

```
2.148E + 06
6.34E + 03
8.0083E - 02
4.444E - 04
```

3. Find the value of each expression below:

A.) SQR(100)
B.) ABS(-25)
C.) SQR(8 * 2)
D.) ABS(-3+-4)
E.) ABS(-3) + ABS(-4)
F.) SQR(4) ^ 4

4. Write a program to find the square, the third power, and fourth power of an inputted value. The output should use sentence formatting.

5. Write a program to find a grade average. Input five scores and find the average. Output all scores as well.

SECTION 1.6 STRING THINGS

Similar to the special mathematical functions, the Apple has special functions designed for string expressions. The next programming problem will lead us into these special functions.

Haven't you been impressed with the way that preppies initial their handbags, sweaters, and paraphernalia. Here's an opportunity to be the ultimate <u>computer</u> <u>prep</u>. Write a program to input first, middle, and last names and then output these as three initials.

This program will require us to select specific letters out of a string. Fortunately, there are several string functions for completing such a task. The first of these is <u>LEFT$</u> function. The LEFT$ function returns a substring of the original string beginning at the leftmost letter. For this function there are two arguments. The first one is the original string; the second argument is the number of characters in the substring. It would look like:

```
LEFT$ (A$, 3)
```

In this example, we get a new string made up of the first 3 letters of the string stored in A$. If A$ = "COMPUTER", then LEFT$ is "COM". LEFT$ (B$, N) returns the first n-characters of the string B$.

The second string function is <u>RIGHT$</u> function. Once again this returns a substring of the original string made up of the rightmost letters. RIGHT$ (A$, N) returns a new string of the rightmost n-characters. RIGHT$ ("COMPUTER", 4) would return the string "UTER".

Even more powerful is the <u>MID$</u> function. This function will allow the programmer to select any sequence of letters located within the original string. The MID$ function needs three arguments. The first one is the original string; the second argument indicates the starting point for the string; the third indicates the number of characters in the new substring. MID$ (A$, 2, 3) yields a substring of A$ consisting of three characters starting at the second character. If A$ = "COMPUTER" then this MID$ (A$, 2, 3) is "OMP". MID$ (A$, 1, 5) is "COMPU". MID$ (A$, 4, 1) is "P". The MID$ function can replace either the LEFT$ or RIGHT$.

With these functions, we are prepared to complete the assignment. Note the print inclusions within the INPUT statements which clearly explain to the keyboard user what needs to be inputted. Also, look carefully at the PRINT statement use of strings to output needed punctuation.

```
10 INPUT "FIRST NAME:"; A$
20 INPUT "MIDDLE NAME:"; B$
30 INPUT "LAST NAME:"; C$
40 PRINT
50 PRINT LEFT$ (A$, 1); "."; SPC(1);
60 PRINT LEFT$ (B$, 1); "."; SPC(1);
70 PRINT LEFT$ (C$, 1); ".";
```

```
FIRST NAME: TIMOTHY
MIDDLE NAME: ISSAC
LAST NAME: MATTHEWS

T. I. M.
```

We've used LEFT$ here, but MID$ will also do the job. Can you rewrite lines 50-70 using the MID$ function? And do you now have an urge to buy a florescent-lime green pair of shorts?

Before leaving strings, there are two other functions related to the manipulation of strings -- LEN and +. The LEN (length) function is used with strings but it is really a numerical function. The LEN function returns a number representing the number of characters in a string. If A$ = "COMPUTER", then LEN (A$) = 8. Note that the only argument is the string for which we wish to find the length. This function can aid with spacing titles and columns and can be a needed feature when using LEFT$, RIGHT$, and MID$.

Suppose we want to change the output of the names to the last letter of each. Instead of T.I.M. we want Y.C.S. In addition, we want to be sophisticated and use the MID$ function. At first glance there is a problem because we have no idea of the length of the names until they are actually inputted. Let's use the LEN function as an aid. Change lines 50-70 to read:

```
50 PRINT MID$(A$,LEN(A$),1);"."SPC(1)
60 PRINT MID$(B$,LEN(B$),1);"."SPC(1)
70 PRINT MID$(C$,LEN(C$),1);"."
```

The result is exactly what we want:

```
Y.C.S.
```

Among the most unusual string functions is CONCATENATION. This simply means joining two strings end to end to produce a new string. The symbol for concatenation is +, not to be confused with numerical addition. If A$ = "WHAT" and B$ = "EVER" then A$ + B$ is "WHATEVER". To demonstrate this function, below is a short program for rhyming:

```
10 INPUT "ANY CONSONANT:"; A$
20 LET B$ = "AT"
30 LET C$ = A$ + B$
40 PRINT C$; SPC(10); "RHYMES WITH SPLAT!"
```

Here C$ is the new string made up of the starting letter and the "at" sound.

```
ANY CONSONANT: B
BAT          RHYMES WITH SPLAT!
```

Manipulation of strings is a powerful tool of Applesoft. String-related functions will allow us to handle verbal information with flexibility. LEFT$, RIGHT$, MID$ give us the ability to select substrings from an original string. LEN function provides us with the means to count the characters of a string. The "+" function allows us to combine strings to form new ones.

EXERCISES 1.6

1. Complete the following table for each of these strings.

	LEFT$(N$,4)	RIGHT$(N$,3)	MID$(N$,2,3)	LEN(N$)
A.) CUCUMBER				
B.) VARIABLE				
C.) BOX				

2. Write a program to take the word STRING and have the anagram RINGS printed out.

3. Write a program which produces the following output. Use the length function as part of the argument for the TAB function.

```
APPLE
 PPLE
  PLE
   LE
    E
```

4. You are so important that your name should be in the middle of everything. So, write a program to input a name and have it centered in the first line of output.

SECTION 1.7 TRACING THE ERROR-FREE ROUTE

"Why checking? I don't need to check, NOT ME!" We hear this from math students every day. There is a good reason!

Rather then continuing to look at a single programming problem, we would like to look at the problems that face every programmer -- ERRORS in the program. Part of a programmer's task is to create error-free solutions. There are two basic programming error types. The first is in the actual grammar of Applesoft. This type is generally referred to as SYNTAX ERRORS. A LET statement with improper addressing; a function without arguments; a variable without a value -- these are examples of syntax error. These can usually be remedied by careful proofreading and by carefully referring to the grammar defined in the reference material.

The second type of error is not a simple one. Yet it is what the challenge of programming is about. A program does not do what is intended. It does not solve the problem. The error may be one of sequencing or perhaps misuse of a variable. Whatever the problem -- the program does not work.

This is not the time to quit. This is the time to put our programming skills to work. Programmers refer to this process as DEBUGGING -- implying that the program is full of bugs, mistakes. Some will be small and hard to find; others nasty to cure. But our job is not done until we have a successful solution.

To help with this debugging, every programmer employs a technique called TRACING. This tracing process goes along with every program we create. It is the ultimate test, the ultimate check, and ultimate bug finder. As the name "tracing" implies, we are going to duplicate the process that the Apple would perform when executing the program. Imagine that the programmer switches on his own RUN button. Tracing is mechanical, but so too is the Apple. One must put himself out of the program and test it as if it were a first time experience.

The tracing of a program consists of two essential parts. First is an active storing of any values assigned to addresses. This means acting as the mailman, placing values into appropriate mailboxes as these assignments occur. To be effective, do not anticipate; simply perform as the Apple would. (But at a little slower pace -- after all, we're just human!)

The second phase of a tracing is the outputting of all information which would appear on the monitor. Anything contained in the PRINT statements must be recorded, and it must be carefully formatted according to punctuation.

The end result of a trace is one of two possibilities. First we come to an error -- a bug. Here we are in a position to do something to change it. The second result is more pleasant -- no errors. And with that you have the self-confidence to know you have accomplished the task.

Below is a sample program, and what follows is a tracing:

```
10 LET R = 6
20 LET Q = 7
30 PRINT R, Q, R * Q
```

In tracing the programmer will execute one statement at a time, and record the necessary task. Our trace is a two-column one. The first represents the table of stored values, the second is the screen output.

STORAGE VALUES　　　　　　OUTPUT

Line 10 is an assignment statement and the tracer executes the assignment by placing it in a mailbox addressed R, the value 6. Line 10 has a similar result. The value 7 is stored in mailbox Q.

```
STORAGE VALUES                    OUTPUT
   R       Q
   6       7
```

Line 30 directs us to the output column. We indicate that 6 (R's value) is placed in the second column. Q's value -- 7 -- is placed in the second column (estimate the second column placement). The final column 3 has in it the value of the product Q * R. Therefore, 42 is in the final position.

```
STORAGE VALUES                    OUTPUT
   R       Q
   6       7          6               7               42
```

The program is error-free, and if the output is what was anticipated, we are satisfied with our solution and our ability.

Despite what algebra students often say about checking, tracing is a satisfying experience. It instills confidence. And it produces successful solutions.

It is worth trying another sample tracing here. This process should be part of each programming process, and will be an important part of this book. Here's one more program from one of our early programming students. Take the time to complete the tracing yourself before reading the explanatory tracing. Practice being mechanical.

```
10 LET B$ = "HELP"
20 LET A$ = "SOS"
30 PRINT "THIS PROGRAM NEEDS SOME";
40 PRINT
50 PRINT "SERIOUS"; B
60 PRINT A$; PLEASE
```

Without knowing exactly what this student intended, we can at least give him what he needs -- some serious help! Line 10 is an assignment of the string HELP in B$. Line 20 assigns SOS to A$. Our mailboxes look like this:

```
STORAGE VALUES                    OUTPUT
   B$      A$
  HELP     SOS
```

Line 30 generates some output. The entire string is printed on one line.

```
STORAGE VALUES                    OUTPUT
   B$      A$
  HELP     SOS           THIS PROGRAM NEEDS SOME
```

Note the trailing punctuation. This probably indicates that there will indeed be further output on this line. But a confusing statement -- a DUMMY print -- is next. This is grammatically correct. It simply adds nothing on the output statement. Since there is now no trailing punctuation, then the Apple skips to the next line when a new print statement occurs. Therefore line 50 is outputted on the next line. We might guess that this is not what was expected. "SERIOUS" was undoubtedly to be outputted following "SOME". Here we can see the value of the trace. Perhaps we might make a quick change -- simply eliminating statement 40. We are now looking at this:

```
10 LET B$ = "HELP"
20 LET A$ = "SOS"
30 PRINT "THIS PROGRAM NEEDS SOME";
50 PRINT "SERIOUS"; B
60 PRINT A$; PLEASE
```

Line 50, printed on the same line, outputs the string "SERIOUS" and then attempts to output a value for B. Unfortunately no such value has been stored. The computer chooses as a default value -- 0 -- for any variable not defined. Certainly "0" is not what our programmer wanted. The string variable B$ is what is called for here. Let's make that change, and we can complete the line output. Line 60 presents problems, too. The computer outputs on the next line since line 50 had no trailing punctuation. The value of A$ is outputted under the first line. Then does the computer output "PLEASE"? -- NOT WITHOUT QUOTATIONS! Instead of the string PLEASE, again we have the Apple searching for the numerical value stored in mailbox PLEASE. Quotations will complete our debugging.

After making all necessary corrections, our trace looks like this:

```
STORAGE VALUES                    OUTPUT
   B$     A$
  HELP    SOS           THIS PROGRAM NEEDS SOME SERIOUS HELP
                        SOS PLEASE
```

A run of this new listed program confirms our corrections. One more program debugged!

```
THIS PROGRAM NEEDS SOME SERIOUS HELP
SOS PLEASE
```

Tracing is essential. No programmer should write a program without checking it through a trace. Both the stored variable values and properly formatted output should be part of this process.

EXERCISES 1.7

1. Trace the following program. Be sure to have the STORED VALUES and the output in correct format.

```
10 LET M = 14
20 LET N = M-10
30 PRINT N
40 LET S = M-N + 6
50 PRINT S
60 LET C = S * -2
70 PRINT C
```

2. Trace the following program. Then write it so that it has the fewest possible statements.

```
10 LET P = 128 + 33
20 LET Q = 146 - 121
30 LET R = 12 * 11
40 PRINT P,Q,R
```

3. Through a trace find the error in this program. Then debug the program.

```
10 LET X = "BUGGED"
20 LET Y$= "DEFINE"
30 LET Z$ = "THIS PROGRAM IS"
40 PRINT Z$ + MID$(Y,1,2) + LEFT$(X$,6)
```

The output should be:

```
THIS PROGRAM IS DEBUGGED
```

4. Each of the individual statements below is incorrect. Rewrite each so that it is an acceptable Applesoft statement.

```
A.)  PRINT LEFT ("DOLLAR",4)
B.)  LET S = O AND B = 6
C.)  PRINT HERE"
D.)  INPUT A;B
E.)  INPUT "HOW MANY CHICKENS CROSSED THE ROAD?":B
```

5. Trace the following program.

```
10 LET X$ = "RADAR"
20 LET B$ = LEFT$(X$,2)
30 LET C$ = RIGHT$(X$,2)
40 PRINT C$ + MID$(X$,3,1) + B$
50 PRINT X$; "IS"; X$; "BACKWARDS"
```

AT THE KEYBOARD SECTION 1B

WHOOPS!

A BOO-BOO! There's no sense learning how to do sophisticated programming before we learn how to correct or edit mistakes. Undoubtedly you will not have typed much at the keyboard before that "WHOOPS" occurs. Our discussion here will be on how to fix up that <u>misstake</u>.

You've begun your typing and line 10 comes out like this.

```
10 LET A$ = HELLO"
```

As an alert typist you pick up your error quickly before hitting the RETURN key. This means that the statement has not been entered and correcting it will be a breeze. You can use the left arrow ( ) and right arrow ( ) keys to move the cursor forward or backwards on a line. Here you need to move the cursor back over to the missing quote and simply type in the new information. As you type, any old information will be erased. If you made a spelling correction, you might simply move over the correct information by pressing the left arrow key until you reached the end of the line. The RETURN key then enters the corrected statement

```
10 LET A$ = "HELLO"
```

Another line typed and you press return. ON NO! You realize another mistake. Now what? You try pressing the right arrow key to move the cursor. No luck. Once you've pressed return, the arrow keys will not move the cursor up. Don't despair, because it can be fixed. Two ways will be discussed here. Later we will learn to correct mistakes in a faster and more sophisticated fashion.

Here is the line to be repaired:

```
30 PRINT HELLO
```

Putting things in simplified terms, the words PRINT and HELLO reside in statement memory location 30. Our goal is to add information to this location. The first strategy is to erase all the information and start over again. This is done by typing the statement numbers where the error resides, 30. Press RETURN. The computer reopens the memory location and erase what is presently there. Since we didn't type anything after 30, the computer places nothing in the location. In effect, the line has been erased. Now 30 can be reopened and restored by retyping the line correctly.

```
30
30 PRINT "HELLO"
```

The second method is essentially the same procedure. But instead of typing 30 two times, once for erasing and once for retyping, we only have to do it once. We can write over the old information. So after typing 30 retype the line with corrections. The computer now has the correct information in line 30.

```
30 PRINT "HELLO"
```

It is worth listing the program to see the results of our correction.

The Apple not only lists the entire program with the correction, but it inserts line 30 in the proper sequence of numbers. You now can see one important reason for line numbers.

Before moving on, let's do a quick review of editing mistakes. If you want to correct a mistake before having hit RETURN, move back to the mistake using the right arrow key. Retype that portion of the proper statement, correctly, or make the change and move to the end of the line using the left arrow key. If you have already entered a statement before realizing the error, retype the line using the same statement number. To delete a line completely, type the statement and press RETURN.

The LIST command plays a major role in the correction process. It allows the programmer to proofread his program. Take advantage of this before you attemp to execute a program.

Two keys that can be used with the LIST command are CONTROL-S and CONTROL-C. If you have a very long list of statements, you may not be able to see all the statements at one time. If the program is being listed, CTRL-C will stop it. You will be able to scan a given portion of the program. CTRL-S, on the other hand, will restart the listing process from the stopping point.

Also of value in doing corrections is a special capability of the LIST command. If a particular statement is in question it is possible to list that one statement. The LIST command followed by a line number will cause only that line to be printed.

```
LIST
10 LET A$ = "APPLE"
20 LET B$ = "TRIAL RUN"
30 PRINT A$.B$
40 PRINT "HOW ABOUT THAT"
50 PRINT A$, "COMPUTER

LIST 50

50 PRINT A$, "COMPUTER
```

To list portions of the program, type LIST followed by the first and last line numbers of the portion desired -- separated by a hyphen.

```
LIST 20-40

20 LET B$ = "TRIAL RUN
30 PRINT A$.B$
40 PRINT "HOW ABOUT THAT"
```

Now we can use our editing techniques to fix line 30. And you can use these "BLUNDER" BEATERS to take the errors out of your programs.

A LITTLE BIT MORE SECTION 1

DETAILS WHICH HELP

In these sections, we will try to explain and demonstrate some of the special touches that you can add to your programs. Some will help simplify your solutions. Others will spice up the programs by demonstrating alternatives. Feel free to experiment with the ideas in each of these sections by employing the techniques in your own programming.

With your brief experience with Applesoft programming we would like to introduce several ideas which might help you in writing a program.

The first of these is a new kind of statement. It is quite unusual becasue it has no effect whatsoever on the program. The REM statement is used within the body of a program to allow the programmer to add any remarks that might be helpful. Everything after the REM command is ignored when the program is run. Nothing is stored or outputted. It is for the writer's use only.

```
10 REM A STANDS FOR INPUT FUNCTION VALUE
```

When might you use a REM statement? It might be used for an introductory remark, to remind the programmer what the program does. It might be used to describe what a variable stands for (as in the above example). It can describe what a particular line in a program does. The REM statement can be used as a reference. Although it will not be used in explanations in the text, it could be a handy tool for your use.

"T.V.", "Mr.", "P.O.", "&" -- all of these are abbreviations. To some degree abbreviations actually replace the original word in our spoken language. Although the idea of abbreviations is to simplify, at times they tend to confuse or cover the true nature of the item. We would like to introduce you to two abbreviations of the Applesoft language. For typing purposes, these can be helpful. We will not use them in the textbook work, for fear they will confuse the true meaning of the statement. You should feel free to use them when they are appropriate.

Our first two Applesoft statements -- LET and PRINT -- can be abbreviated. The variable assignment statement LET can be written without the word "LET" to introduce the assignment. The statement 10 LET X = Y * 2 can be written:

```
10 X = Y * 2
```

The computer understands that without the introductory word "LET", what follows the line number is a variable name. And this is only true with a LET statement.

```
20 LET B = SQR(12) becomes...   20 B = SQR(12)
30 LET C$ = "ABBREVIATION" becomes...
            30 C$ = "ABBREVIATION"
```

The other abbreviation is for the PRINT statement. The question mark following the line number is understood by the computer as PRINT. The line 10 PRINT "HELLO" has the exact same output as the line:

```
10 ? "HELLO"
```

In fact, if you listed the program, the question mark is replaced by the word PRINT. The advantage of the question mark abbreviation is the time saved when a program contains many PRINT statements.

Both abbreviations can save time, but do not let them confuse you as to the function of each statement. As with all abbreviations, make good use of them.

REVIEW EXERCISES

CHAPTER ONE

1. For each problem below, give the output for the indicated Applesoft function.

```
A.)  PRINT ABS(14-19)
B.)  PRINT RIGHT$("NUMBER 1",5)
C.)  PRINT SQR(2.5 ^ 2)
D.)  PRINT MID$("SLICE",3,3)
E.)  PRINT 3 * 4 + 2 ^ 3
F.)  PRINT LEFT$("BOX",1) + RIGHT$("BOX",1)
G.)  PRINT MID$("PROGRAM",3,LEN("PROGRAM")-3)
H.)  PRINT 2 * (6-4 + 1)
I.)  PRINT ABS(4 * -6/3 - 2)
J.)  PRINT LEN("HAPPY DAYS")/2
```

2. Write a program to input a name and to print it out so that it is moved over from the margin as many spaces as there are letters in the name.

3. Write a program to input two numbers and output the sum, difference, product, and quotient of their squares. No other values except the two original numbers are to be stored. Output the two numbers and then in column form, with headings: the sum, difference, product, and quotient.

4. Trace the following program with a table of STORED VALUES and the properly formatted OUTPUT. Make all necessary corrections in syntax.

```
10 LET A = 10
20 LET B = A÷2 + 3
30 LET Q = "?"
40 LET P$ = "THIS"
50 LET A$ = IS RIGHT
60 PRINT A:B,A - B
70 PRINT A; "-"; B = A - B
80 PRINT LEFT(A$,2); SPC(2); P$; TAB(2); RIGHT$(A$,5); Q$
```

5. Write a program to input your name and to have it outputted twice, once horizontally and once vertically.

```
JENNIFER
E
N
N
I
F
E
R
```

CHAPTER 2

WHAT IF . . . ?

Section 2.1 Following Your Nose
Section 2.2 Programming Jumps
Section 2.3 The First R -- Teaching the Apple To Read
At The Keyboard: Disk In Use
Section 2.4 Decisions, Decisions, Decisions
Section 2.5 We Can Solve All Of Those Errors, If...
Section 2.6 No If's And's Or But's
A Little Bit More: Spicing Up The Print Statement
Review Exercises

FOLLOW YOUR NOSE

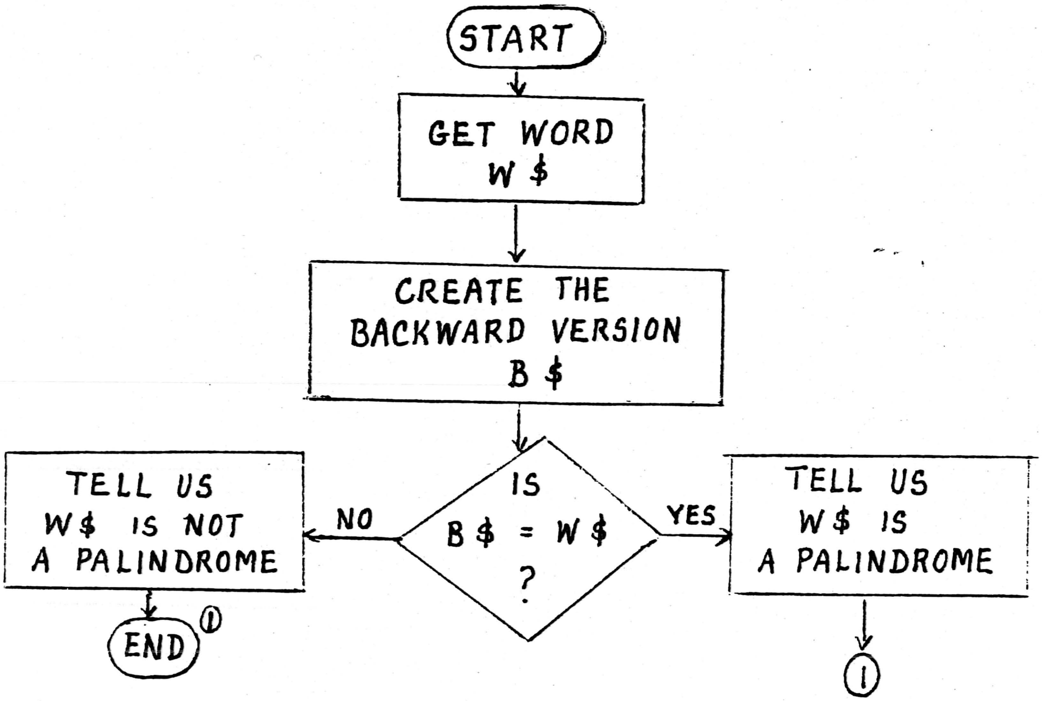
With some Applesoft under our belts, we are prepared to begin to look at general techniques of programming. And to help with the ever-growing complexities, we would like to offer another aid in programming -- the use of a flowchart. A flowchart is a model for step-by-step instruction giving. It is a mathematical problem-solving recipe, complete with symbols.

A programmer uses a flowchart as an artist would use a sketch pad. A flowchart will represent an initial sketch of the problem's solution. It includes all the instructions and techniques, but it is keyed to no specific language. A programmer uses his flowchart then to translate the solution into the required language -- computerese.

Throughout our discussion of Applesoft, we will use flowcharts to develop our solutions. As a novice programmer, the flowchart should be part of your solving process for each task.

"Able was I ere I saw Elba." So spoke Napoleon as he stood on the bridge of his exile ship. These words have become infamous however, for the sentence is a PALINDROME. A palindrome ia a word or phrase which is the same backwards as forwards. "Radar", "madam",and "Otto" are examples of palindromes. Our problem here is to have the Apple determine whether a word is a palindrome.

Below is a flowchart which has been constructed to solve the problem:


START
GET WORD W $
CREATE THE BACKWARD VERSION B $
IS B $ = W $?
NO
YES
TELL US W$ IS NOT A PALINDROME
TELL US W$ IS A PALINDROME
END
①
①

In a flowchart, all instructions are placed in rectangles. Here the first instruction is "GET THE WORD". Note that we are not concerned with using BASIC language, but merely in the idea of inputting the possible palindrome. Note the use of a variable, A$, inorder to refer later to the word. Arrows are used throughout the flowchart to indicate the direction of movement. These arrows create paths. An arrow directs us to an instruction to create the backward version of the word, B$. This would obviously mean a concatenation of A$.

It is possible within a solution to ask questions. These questions must be in a form of a YES-NO decision. The Apple is capable of making choices, but only YES or NO. A diamond box is used for all decision-making questions. And from each decision there will necessarily be two paths -- a YES path and a NO path. Our question, "IS A$ = B$?", is really asking if the word is a palindrome. The YES path leads us to a set of instructions for outputting the conclusion: A$ IS A PALINDROME. The NO path leads us to the opposite conclusion.

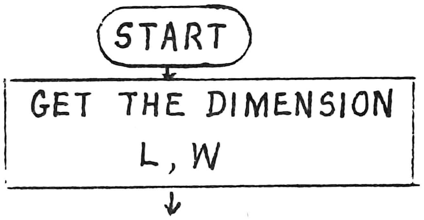
The other symbol, used in the flowchart is a circle. In our program it is used three different times, and for different reasons. In every flowchart we will have a START and a STOP. These are to simply indicate a beginning and end to the solution. They initialize and terminate all paths. The other use of the circle is for jumps in pathways. After the instruction, "TELL US A$ IS A PALINDROME", the arrow leads to a small circle ① . This indicates that the path continues at the instruction labelled ① . Therefore in the flowchart the YES path converges back into the NO path at the instruction -- STOP. This use of a circle is called a connector, and is a way to indicate the direction of movement. It is used to replace the arrow so that the flowchart does not become confused with overlapping or intersecting arrows. Whenever there is a chance for one to confuse the movement, use the connector. Be sure not to forget to label the instruction box where the movement continues, with the same number.

You can see how the flowchart is used to organize the solution. The only task remaining is to translate the flowchart into the desired language. This is done by translating each instruction, decision, or connector into Applesoft. You can see how the flowchart is the essential "thinking" part of programming. And you can see why flowcharting will be an essential part of our work.

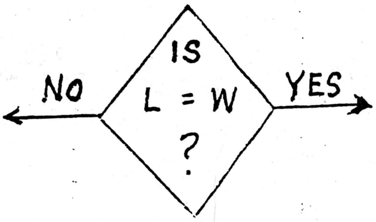
Just to help us get oriented to the symbols and techniques of flowcharting, we would like to construct a flowchart solution together. This time our problem to flowchart is a "frame-up.

The Frame Factory is running a special deal on framing supplies. The price of any rectangular frame is $.12 times the perimeter. And if it is square, the Frame Factory will give you $2.00 off the price. Construct a flowchart to allow a person to input the dimensions of the picture and to have the Apple calculate the cost.

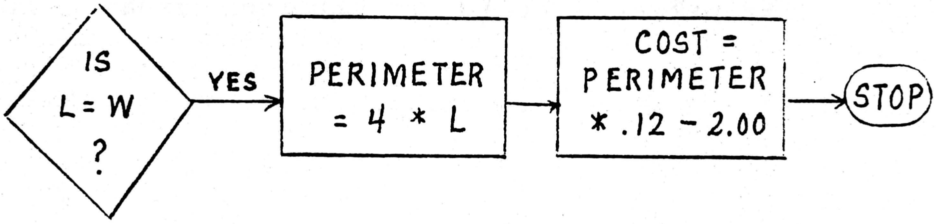
Our flowchart, like all others, begins with a START circle. An arrow will take us to our first instruction. To begin we must have the length and width of the picture. Check our flowchart beginning:



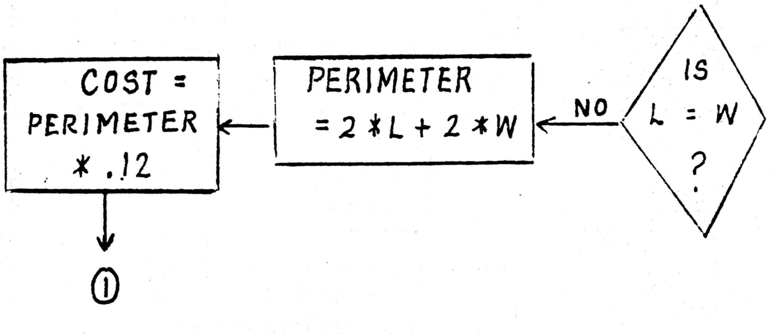
Again note the use of a variables, L and W, in order to refer to the dimensions later in the flowchart. We are ready for the decision: Is this a square? In our program that is equivalent to asking if L = W. An as with all questions, there must be exactly two paths generated by the diamond-shaped decision. Note the use of the "YES" and "NO" to indicate the two paths.



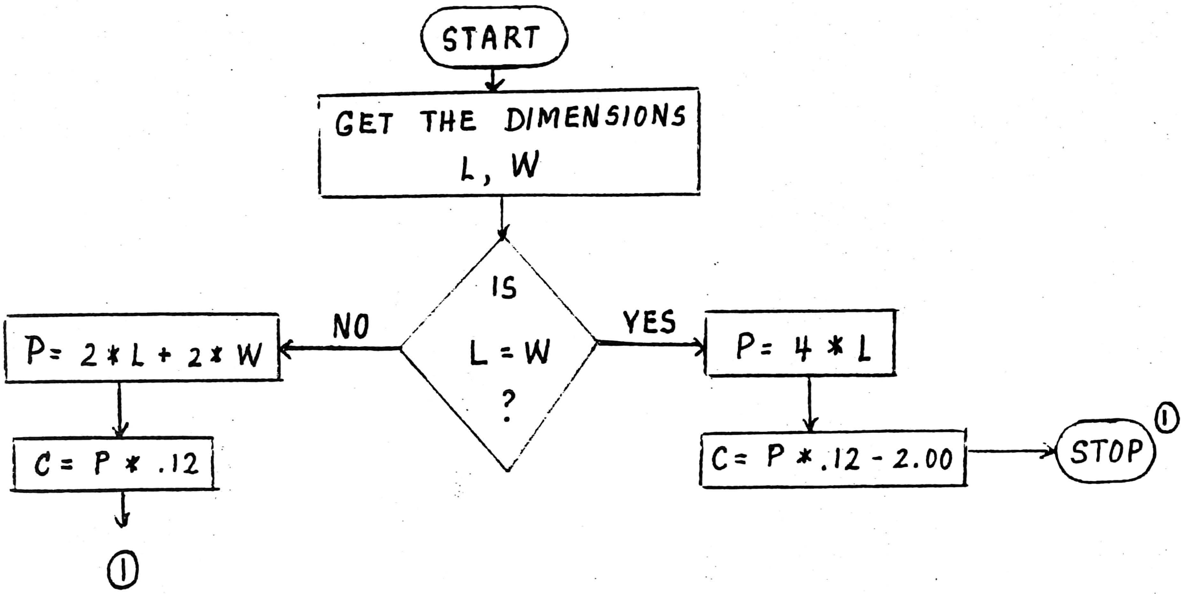
The YES path is the direction we move if the frame is square, and our next command should be to find its perimeter. The YES path is completed by finding the price and deducting the discount. The arrow moves to the STOP circle. It looks like this:


IS
L = W
?
YES
PERIMETER
= 4 * L
COST =
PERIMETER
* .12 - 2.00
STOP

The NO path, on the otherhand, will lead to the computing and outputting of the cost of a rectangular frame.


COST =
PERIMETER
* .12
PERIMETER
= 2 * L + 2 * W
NO
IS
L = W
?
①

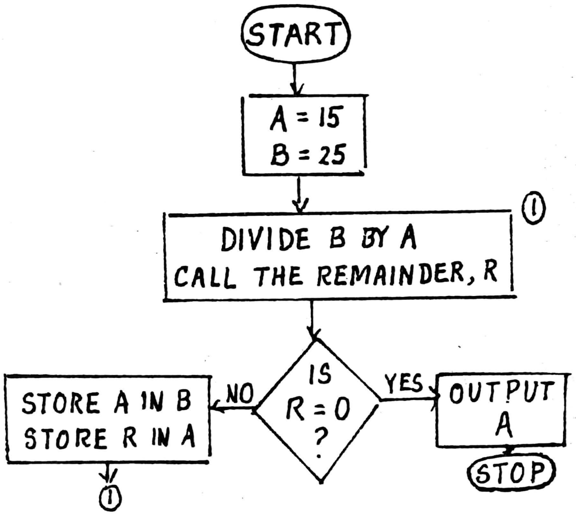
Since both paths must end at the same location, the NO path output instruction must also converge at this STOP circle. Here a long arrow or the connector might be used. Check the notation here when the connector is used:


START
GET THE DIMENSIONS L, W
IS L = W ?
NO
YES
P= 2*L + 2*W
C = P * .12
P= 4*L
C= P*.12 - 2.00
STOP
①
①

Another problem solved. And by the time Chapter III approaches we will be ready to translate this flowchart into an Applesoft program. Be a successful programmer and use the flowchart to help you solve your problems. It is not a time-wasting effort -- it is a place to create without the restraints of programming language syntax. Use flowcharts!

EXERCISES 2.1

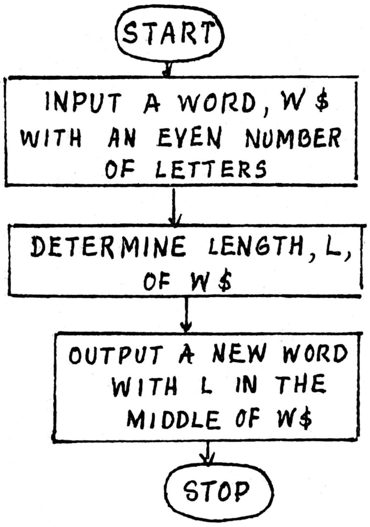
1. Below is a mystery flowchart. Follow the flowchart through the correct paths to determine the final output. See if you can guess what problem the flowchart solves.


START
A = 15
B = 25
DIVIDE B BY A
CALL THE REMAINDER, R
①
IS R = 0 ?
NO
YES
STORE A IN B
STORE R IN A
①
OUTPUT A
STOP

2. Construct a flowchart to solve this problem. Input a height in inches and output the measurement in feet and inches. For example, 150 inches equals 12 feet 6 inches. Include at least one decision box in your flowchart.

3. For us, telling if a number is positive, negative, or zero is easy. For the Apple it requires several questions. Create a flowchart which inputs a number and determines whether it is positive, negative, or zero.

4. Translate the following flowchart to an Applesoft program. Each box is a new statement.


START
INPUT A WORD, W$ WITH AN EVEN NUMBER OF LETTERS
DETERMINE LENGTH, L, OF W$
OUTPUT A NEW WORD WITH L IN THE MIDDLE OF W$
STOP

Ex:

COMPUTER...becomes COMP8UTER

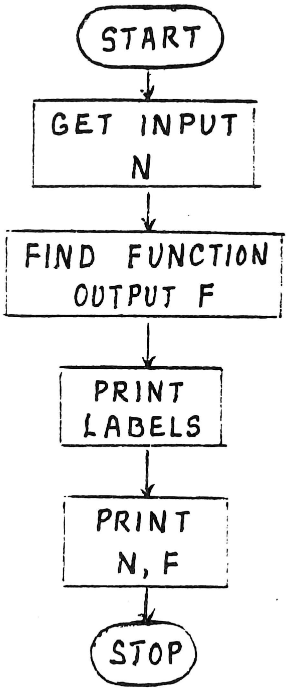
SECTION 2.2 PROGRAMMING JUMPS

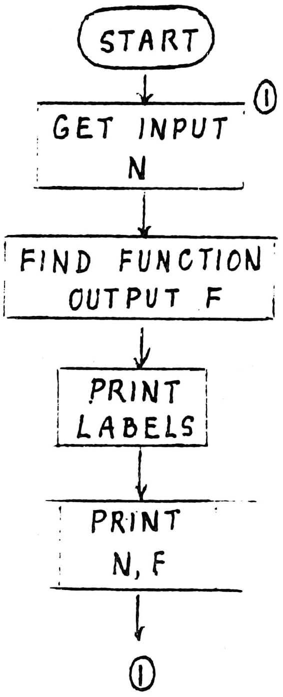
Up to now, our work with the function machine program has been severely limited. Our final programming solution from Section 1.4 prohibits us from inputting more than one value each time the program is executed.

```
10 INPUT "INPUT A FUNCTION VALUE: "; N
20 LET F = 2 * N + 1
30 PRINT FUNCTION MACHINE -- F(X) = 2X + 1
40 PRINT
50 PRINT "INPUT", "OUTPUT"
60 PRINT N, F
```

Although the output is attractive, there is little beauty in such a handicap. We would like to modify this feature. We want a program that allows the user to continue inputting values into the function machine.

The function machine flowchart is shown on the left.





Simply stated, we wish to replace the STOP circle with an arrow which reinstates us into the program. That is, we wish to continue at a previous instruction. That instruction will be "GET INPUT VALUE N". The flowchart on the right has been altered to send us back to the first instruction. Each time the output instruction is carried out, the program jumps back to the "GET INPUT VALUE" instruction. In programming, this jump is called an UNCONDITIONAL TRANSFER. The unconditional transfer is a program jump -- no matter the conditions. In Applesoft, the statement used looks like this:

```
70 GO TO 10
```

The jump command is GO TO, followed by the statement number line to which the program will transfer. Our program is now:

```
10 INPUT "INPUT FUNCTION VALUE: "; N
20 LET F = 2 * N + 1
30 PRINT "FUNCTION MACHINE -- F(X) = 2X + 1"
40 PRINT
50 PRINT "INPUT", "OUTPUT"
60 PRINT N, F
65 PRINT
70 GO TO 10
```

We have inserted line 65 -- a dummy PRINT statement -- to separate the new input. When run the program output is:

```
INPUT FUNCTION VALUE : 15
FUNCTION MACHINE -- F(X) = 2X + 1

INPUT          OUTPUT
15             31

INPUT FUNCTION VALUE : 25
FUNCTION MACHINE -- F(X) = 2X + 1

INPUT          OUTPUT
25             51
```

The first time statement 10 was executed the keyboard operator inputted the value 15. The properly headed output prints 15 and 31. Immediately the program transfers to statement 10, and the keyboard operator is once again requested to input a number. This time 25 resulted in the second part of the output -- 25 and 51. And again the button pusher is asked to do some work.

As you might be thinking, this program does not stop! Our flowchart should have been a first clue. Look back and you will see that there is no STOP circle. Hence there is no end to our jumps. In computerese, we have created an endless program. And the Apple, being a mechanic at heart, will continue on endlessly -- without a question or comment.

Such a flaw is what learning programming is all about. To fix this flaw within a program will be a goal in our later work. Fortunately, a keyboard operator can stop the execution of a program. Once a program is being run, it is possible to "break" into the sequence of instruction. Pushing CONTROL-C stops the program execution. Remember when using control characters -- hold down the control key (CTRL) while pushing the "C" key. The computer will respond by stopping and by indicating where the action terminated:

```
BREAK IN 10
```

Therefore, CTRL-C gets us out of trouble. But this is something we certainly need to work towards in our programming.

The unconditional transfer -- GO TO -- allows the programmer to make jumps in the sequencing of statements. This gives us the ability to do repetitions or to skip unwanted parts of a program. As parentheses override the normal order of operations in arithmetic, so the unconditional transfer statement overrides the Apple's impulse to proceed in ascending order through our sequence of instructions. Once again we have gained control.

Speaking of control, at this point in our programming there is always one student who enjoys an endless prank. Trace through this program using the unconditional transfer.

```
10 PRINT "HAVE YOU EVER SEEN THE"
15 PRINT "PERPETUAL MOTION MACHINE?"
20 PRINT
30 INPUT A$
40 PRINT
50 GO TO 10
```

It's more fun to be on the other side of this program -- <u>writing it</u>.

EXERCISES 2.2

1. What statement must be added to the following program to allow the user to repeat the input process?

```
10 PRINT "FUNCTION MACHINE"
20 PRINT
30 INPUT "INPUT A NUMBER?";N
40 PRINT
50 PRINT N, N ^ 2 + 5
```

2. Do you know how to tell how close lightning strikes? Just count the pause in seconds between the time you see lightning and hear the thunder, and divide by five. That quotient is the number of miles away the lightning is. Construct a flowchart which allows the user to input the pause time and which computes the distance. Make provisions to repeat the process.

3. Trace through this program until you reach line 50 for the third time. Then explain the flaw in the program and suggest a remedy.

```
10 PRINT "SOMETIMES"
20 GO TO 50
30 PRINT "YOUR DESTINATION"
40 GO TO 10
50 PRINT "YOU NEVER REACH"
60 GO TO 20
```

4. Every fraction can be represented as a decimal by dividing the numerator by the denominator. Write a program to allow the user to input the numerator and denominator and to convert the fraction to a decimal. Again allow for multiple input. The output should look like:

```
1/2 = .5
```

5. Each grading period, the math students have three test scores. Write a program to input the student's name and his/her three grades. Then compute the average for the student. Allow the teacher to enter as many sets of grades as necessary. The output should say:

```
STUDENT          AVERAGE

D. FELT            92.6
```

THE FIRST R -- TEACHING THE APPLE TO READ

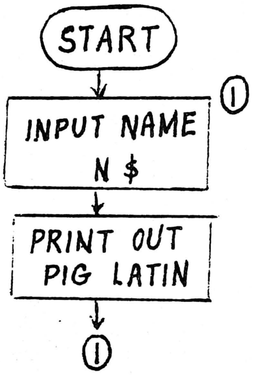
Making variable assignments has been used in each and every program we have written. Two techniques have been used. First is the LET statement, which allows for a single assignment. The other is the INPUT statement, which allows for multiple assignments, but only at the keyboard. Our programming problem will look at yet another way to assign a value to a variable.

As a childhood diversion, it was always fun to play games with words. "EARNINGLAY ASICBAY" -- that's Pig Latin for "LEARNING BASIC". The Pig Latin language takes the first letter off of a word, and then adds "ay" to it. That becomes the last syllable of the new Pig Latin word. LET becomes ETLAY. PRINT becomes RINTPAY. Write a program which will input three-letter names into a function machine and have the names outputted in Pig Latin. For this program however, we want several names inputted, as part of the program.

To accomplish that task at this point, our program would look like a series of duplicate programs. A new LET statement would be necessary for each new assignment.

```
10 LET A$ = "TIM"
20 PRINT MID$ (A$, 2, 2) + MID$ (A$, 1, 1) + "AY"
30 LET B$ = "JAN"
40 PRINT MID$ (B$, 2, 2) + MID$ (B$, 1, 1) + "AY"
50 LET C$ = "LEE"
60 PRINT MID$ (C$, 2, 2) + MID$ (C$, 1, 1) + "AY"
```

What we need is a transfer to the assignment location. But at this point to accomplish this requires the INPUT statement. In our flowchart the way to translate the instruction "GET A$" was by the use of an INPUT statement. This requires, however, that a keyboard operator is ready to respond to the input requests.



There is an alternative to this statement that meets the requirement. The alternative consists of the two statements "READ" and "DATA". The READ statement might look like:

```
10 READ A$
```

This statement is an assignment one, however. Upon executing this, the computer sets up a mailbox addressed -- A$. But as opposed to asking for a keyboard input, the computer goes elsewhere for the value. The value it searches for needs to be located in the DATA statement. The DATA statement is a non-executed statement; it can be located anywhere in the program. If we were to trace this, the DATA statement does not require us to do anything. It is merely a storage location for any and all values which we wish to store in A$.

```
15 DATA "TIM", "JAN", "LEE"
```

When the Apple reaches the READ statement, it scans the program for the first DATA statement and selects from it the first item in the list. That value is assigned to A$. If the Apple returns to that READ statement again, the second item from the DATA list is assigned to A$. The first value in A$ is then replaced.

Our program might look like this:

```
10 READ A$
15 DATA "TIM", "JAN", "LEE"
20 PRINT MID$ (A$, 2, 2) + MID$ (A$, 1, 1) + "AY"
30 GO TO 10
```

Let's trace the program. In statement 10, the computer stores in A$, the first data item "TIM". Statement 20 performs the MID$ function to put together the three letters from last to first. Also that word is outputted. Statement 30 transfers us to 10 and the Apple searches for and finds a new value for A$. "TIM" is replaced by "JAN" in mailbox A$. Imagine if you will that there is a pointer within the DATA statement. As each piece of data is selected, the pointer moves to the next piece. In this way the Apple is aware of which item is to be used next. The third time around, the word "LEE" is selected and "EELAY" is printed out. Upon returning to the READ statement a fourth time, a difficulty arises. The Apple searches for the DATA statement but there is no fourth item in the list. If there were another DATA statement, the pointer would have moved to the first item in that next DATA statement. Since there is no other item, the computer senses this as an error. The program execution halts, and the Apple responds with an error message. The complete tracing would look like this:

STORAGE VALUE A$	OUTPUT
TIM	IMTAY
JAN	ANJAY
LEE	EELAY
	OUT OF DATA ERROR

This indicates that there is no more data available to be stored in A$. Although it is never a good idea to end with an error, this is an acceptable way for us to end such a program at this point.

The READ-DATA statements are a useful tool for making multiple assignments. They are interchangeable with the INPUT statement. Obviously the way in which the values are to be assigned -- at the keyboard or within a DATA statement -- will determine which should be used.

As with INPUT statements, it is possible to read in assignments for more than one variable. A second READ-DATA example will demonstrate.

At your Basic and Robbins Three Flavors Ice Cream Parlor, you wish to program your cash register to keep track of sales. The program should allow for the customer's name and the number of single, double, and triple dip cones that were purchased. The output should represent the final bill for the customer. Single dip cones cost $.80, double dips $1.20. and triple dips $1.50.

In this program, four pieces of information need to be assigned for each customer -- NAME, SINGLE DIPS, DOUBLE DIPS, TRIPLE DIPS. Try your hand at writing the READ statement prior to looking at ours:

```
10 READ N$, S, D, T
```

The name is a string variable as compared to the numerical variables for the different cone types. The items of the DATA statement must be arranged exactly as they are to be assigned. The first piece of data must be a string; the next three need to be numbers. The fourth piece would need to begin the process again. It might look like:

```
30 DATA "SMITH", 2, 3, 0, "JONES", 1, 1, 1
35 DATA "BROWN", 4, 5, 6
```

When the READ statement is executed, N$ receives as its value the first data item, S receives the next, and so on. After the third time around the table of STORAGE VALUES would look like this:

STORAGE VALUES

N$	S	D	T
SMITH	2	3	0
JONES	1	1	1
BROWN	4	5	6

See if on your own you can complete the problem by writing the entire program. You will need to calculate the final bill for each customer. Be sure to include a good creative output statement. We used a sentence format; you might wish to use a column formatted output.

```
5 PRINT "BASIC AND ROBBINS"
10 PRINT "ICE CREAM STORE"
15 PRINT
20 READ N$, S, D, T
30 DATA "SMITH", 2, 3, 0, "JONES", 1, 1, 1
35 DATA "BROWN", 4, 5, 6
40 LET T = S * .8 + D * 1.2 + T * 1.5
50 PRINT
60 PRINT N$;"'S BILL IS FOR $";T;"."
70 PRINT
80 GO TO 20
```

We obviously needed to use the unconditional transfer here. But be careful to transfer to the READ statement rather than to the non-executed DATA statement. And remember, despite having these new tools, we still need to learn to avoid that OUT OF DATA ERROR.

Now get to the exercises -- before they melt!

EXERCISES 2.3

1. The Apple can easily be made to generate table output. Create a table of the squares of each of the following pieces of data: 2, 6.7, -13.21, 0, 24, 98, -71.2

2. Trace the following program with a table of STORED VALUES and the properly formatted OUTPUT.

```
10 PRINT "X", "Y", "Z"
20 READ X,Y,Z
30 PRINT X,Y,Z
40 PRINT
50 PRINT 2 + X * Y ^ 2 - 8
60 PRINT
70 GO TO 10
80 DATA 0,2,3,4,-1,8,16,1,1
```

Do not forget to use the correct order of operations.

3. Ted Williams is the last major league baseball player who ended a year with a batting average over .400. Averages are computed by dividing the number of hits by the number of times at bat. Write a program for computing the batting average of five of your favorite baseball players. The output should look like this:

```
 NAME      HITS      AT BATS      AVERAGE

CAREW       20          64           .375
```

4. We sure hope this wasn't your program! Help debug this program by correcting each statement. Be on your toes!

```
10 READ A;BS
20 DATA "JACK", 6, "JILL"; 7, "HILL", 16,"PAIL"
30 PRINT BS "IS" A YEARS OLD
40 GO TO 20
```

DISK IN USE

The disk drive system of the computer can be compared to a record player. However, the disk drive can not only "play back" programs, but it can also "record" them for future use. This section is devoted to the Disk Operating System, or DOS, and all of the associated commands necessary to save, load, run and catalog programs.

The first step in using a disk drive is to prepare a blank disk, or to initialize it. The Apple uses a "floppy" disk for storing information. It is called floppy because it is bendable. But they are very fragile. If you bend them, expose them to hot temperatures, touch the surface visible through the window at the bottom of the protective covering, or expose them to a magnetic field, the disk may be destroyed. Be careful.

Though all microprocessors use the same type of disk, they store the information in different ways. Some store the information in concentric circles very much like a phonograph record; some store in pie shaped slices called sectors; some, like the Apple, use a combination of both. Initialization of a disk sets up invisible divisions on its surface as indicated by the mechanics of the computer, so that information can be stored and accessed quickly.

The first step to initialize a disk is to insert one that has already been initialized, the "System Master" disk, for instance. This is done by first opening the gate on the front of the disk drive. Next, holding the disk so that the little notch cut in the side is to the left and the oval window is towards the drive, carefully insert the disk into the drive. Finally, close the gate. After completing this process, find the power switch on the back left hand corner of the computer. Turn the computer on and wait for the red "IN USE" light to turn off, a process that should take about five seconds. What we have just done is booted the disk, a fancy way of saying turned on the disk drive. In fact, we have just done a cold boot, a term to be discussed a little later. The next step is to replace the original disk with your blank disk, the one to be initialized. Now, type the command NEW to erase any data in the computer's memory, and then type a program that you want to be run each time the disk is booted. For example,

```
10 PRINT "DISK CREATED BY APPLE SLICE"
20 PRINT "ON SEPTEMBER 3, 1982
30 END
```

Finally, type the disk command INIT HELLO. INIT stands for initialize and the word HELLO is simply the name of your program. It should be noted that we could use any label that begins with a letter to name the program. This is a greeting program and is executed each time the disk is booted. HELLO is a logical choice for such a program.

After typing INIT HELLO and pressing RETURN, the disk drive will spin for about one minute as indicated by the IN USE light. During this period, a number of different operations are taking place. The first completely erases the disk and for this reason you only initialize a disk once. The second operation divides the disk into thirty-five separate sections called tracks. Each track is divided into sixteen different sectors. Information is stored on one sector at a time until the disk is full.

Now that we have initialized the disk we are ready to investigate how to save programs. To save a program, you must give it a name, just like each book in a library has a title. The name of a program can be any that you desire except that it must begin with a letter. If you give your program the same name as a program that is already on the disk, the old program will be erased, and your new one put in its place. The SAVE command is used to finish the process. Type the command:

```
SAVE (PROGRAM NAME)
For example,
SAVE DISK DEMO
or
SAVE EXAMPLE #1
```

Press RETURN and wait until the IN USE light turns off. If nothing happens, press the key marked RESET and try again. Incidently, if you ever have a problem, pressing RESET will allow you to start the step over.

To verify that your program has been saved, type the disk command CATALOG and press RETURN. You should observe the disk drive spinning and in a moment, the names of all the programs on the disk will appear on the screen. Below is a copy of one such catalog:

```
]CATALOG

DISK VOLUME 254

*A 006 HELLO
*I 018 ANIMALS
*T 003 APPLE PROMS
*I 006 APPLESOFT
*I 026 APPLEVISION
*I 017 BIORHYTHM
*B 010 BOOT13
*A 006 BRIAN'S THEME
*B 003 CHAIN
*I 009 COLOR DEMO
*A 009 COLOR DEMOSOFT
*I 009 COPY
*B 003 COPY.OBJ0
*A 009 COPYA
*A 010 EXEC DEMO
*B 020 FID
```

In the catalog each program saved is listed. Also indicated is the language the program is written in and the sector space used. If there happens to be more programs on the disk than will fit on the screen, press any key except RESET to see the remaining programs.

In order to have access to a program saved on the disk, we must load the program from the drive to the Apple itself. The command to do this is LOAD. Type this command followed by the exact cataloged name.

```
LOAD(PROGRAM NAME)
For example,
LOAD DISK DEMO
or
LOAD EXAMPLE #1
```

After pressing return, the program can be listed (LIST) to see its contents or it can be run by typing the command RUN.

It is possible to bypass the loading process and have the program executed immediately. Type the RUN command followed by the program name.

```
RUN (PROGRAM NAME)
For example,
RUN DISK DEMO
RUN EXAMPLE #1
```

When executed, the following things occur. First, the computer finds the program in the catalog. Secondly, it loads the program into the Apple memory. Finally, the computer executes the program.

Another topic needs to be discussed before moving on -- the difference between a cold boot and a warm boot. A cold boot, you may remember, is simply inserting an initialized disk into the drive before turning on the computer. When the power is turned on, the drive is automatically turned on and the disk ready for access. What happens, however, if you press the RESET button immediately after turning on the computer? You will notice that the disk drive turns off and typing any of the disk commands will result in SYNTAX ERROR. You can use the computer as you would normally but you cannot save your program onto a disk. You would use this procedure, for instance, if you did not have a disk available to you when using the computer.

If halfway through your programming you decide you want to start saving your programs, you can do a warm boot (the term warm is derived from the fact that the computer has been turned on prior to this procedure and is thus warm). Type PR#6, the abbreviation for port number 6. There are actually seven slots or ports inside th computer and the number 6 indicates that the disk drive is plugged into the sixth one, its usual location. If for some reason nothing happens after pressing RETURN, press RESET and try again. There are two warnings to be heeded. If you press RESET before allowing the IN USE light to turn off, you cannot save your program. Also when you type PR#6 to access the disk drive, it first erases all programs in memory! So don't start writing programs that you want to save until a disk has been booted.

The disk drive allows us to store and access a great deal of information. Used carefully, the DOS expands the Apple's memory capabilities geometrically.

It is time to ask some serious questions! Not of yourself, but of the Apple. That's right -- it's time to consider how the Apple handles the decision-making process. Up to this point, our programs have not encountered a question box.

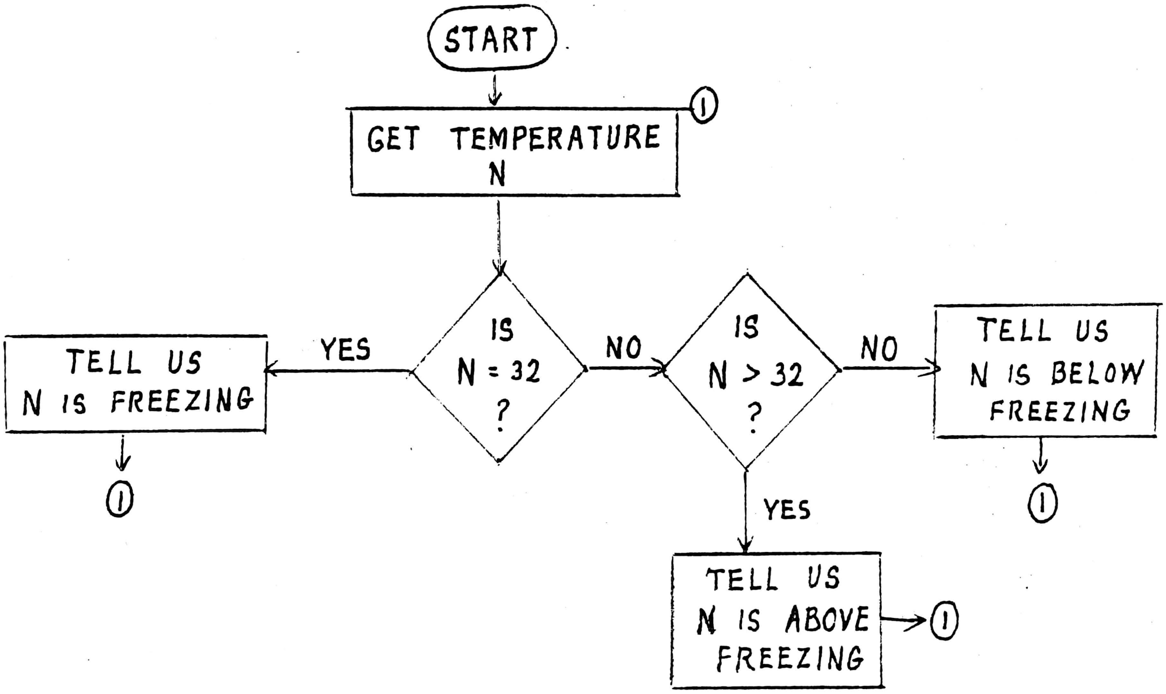
Our programming problem is to have the Apple act as a temperature gauge. Allow for a Fahrenheit temperature to be read in and determine whether that temperature is freezing, above freezing, or below freezing.

Before we try our hand at constructing the flowchart, let's think out our solution. We want to get an input. Then we will need to ask the question: Is the temperature freezing? Actually we will be comparing the number to 32. If the temperature is 32, then we want the output to say so. If not, we will need to ask another question: Is the number greater than 32? If so, the output will say that the temperature is above freezing. If the answer is no, do we need to ask another question? Definitely not, because we have eliminated the other possibilities and the temperature must be below freezing. After each output we will want to return for additional inputs.

Now from our little discussion, we would like to try our hand at constructing the flowchart. Make sure each decision box has only one YES path and one NO path. Also the Apple uses the same symbols we use for the relationships greater than (>) and less than (<).

Don't look at our flowchart until you've tried one yourself! Think!

Here's our flowchart. It won't necessarily be yours, but it is the one we will use to solve the program.

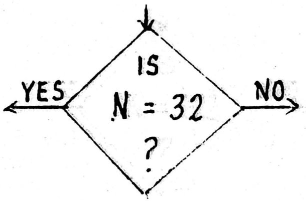


Note the two question boxes, each with a YES and NO path. Also you will see that all the paths return to the instruction -- "GET N".

We are going to begin translating this flowchart into readable Applesoft. Instructing the Apple to "GET N" could be translated as either an INPUT or READ statement. With our recent work with READ statements, that will be our choice here.

```
10 READ N
20 DATA 12, 57, -6, 32
```

In the flowchart, we have come to our first decision diamond.



How do we go about translating this question? All decisions will be translated into a single form -- CONDITIONAL TRANSFER. Conditional, because a certain STATEMENT must be satisfied before the transfer can take place. The conditional transfer consists of two parts. The first part is a conditional statement which can be either true or false. The second part indicates where the program should jump to, if the conditional sentence is true. In this example, the statement might look like this:

```
IF N = 32 THEN 120
```

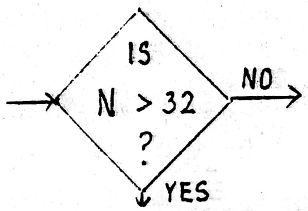
When the Apple executes this, it decides whether or not the statement is true. If N = 32 is true the program transfers to line 120. If the statement is false however, the Apple skips sequentially onto the next statement. Our first decision translates to:

```
30 IF N = 32 THEN ______
```

The blank simply indicates that we will later complete the statement number.

Remember that if the statement N = 32 is false, the computer proceeds to the next statement, another decision. Therefore when translating from flowchart to program, we will always follow the complete NO path throughout the entire flowchart. As humans, we always follow our nose! Only when this path is complete will we return to include the YES paths.

The NO path leads us to this question:



Try to write another IF-THEN statement for this box. It should look like this:

```
40 IF N > 32 THEN ______
```

Again, we cannot fill in the statement number yet. And once again we will follow the NO path. What is the next instruction?

TELL US N
IS BELOW FREEZING
①

By now this should not present a problem. But be creative! The final instruction in the NO path is the connector box. Our unconditional transfer -- GO TO -- will help us complete the NO path. The program through the NO path completion consists of these six statements.

```
10 READ N
20 DATA 12, 57, -6, 32
30 IF N = THEN ____
40 IF N>32 THEN ____
50 PRINT N; " DEGREES FAHRENHEIT IS BELOW FREEZING."
60 GO TO 10
```

The program is certainly not complete. Lines 30 and 40 are incomplete, because we have not taken care of the YES paths. Return to the first IF-THEN statement: 30 IF N = 32 THEN _____. We can now specify the transfer statement.

```
30 IF N = 32 THEN 70
```

Complete this YES path with PRINT and GO TO statements. Then do the same for the other YES path:

```
40 IF N > 32 THEN 90
```

Here is the complete program with the output from the execution.

```
10 READ N
20 DATA 12, 57, -6, 32
30 IF N = 32 THEN 70
40 IF N > 32 THEN 90
50 PRINT N; " DEGREES FAHRENHEIT IS BELOW FREEZING."
60 GO TO 10
70 PRINT N; " DEGREES FAHRENHEIT IS  FREEZING."
80 GO TO 10
90 PRINT N; " DEGREES FAHRENHEIT IS ABOVE FREEZING."
100 GO TO 10
```

```
12 DEGREES FAHRENHEIT IS BELOW FREEZING.
57 DEGREES FAHRENHEIT IS ABOVE FREEZING.
-6 DEGREES FAHRENHEIT IS BELOW FREEZING
32 DEGREES FAHRENHEIT IS FREEZING.
OUT OF DATA ERROR.
```

The technique of using conditional transfers to separate the program into various pathways is called <u>BRANCHING</u>. The power to make decisions places the computer well above the hand-held calculator. And the ability to program with decisions increases geometrically our solution powers.

All conditional transfers use the IF-THEN form. The statement after the "IF" must be a mathematical sentence. The Apple has to be able to ascertain the truth value of this sentence. Under true conditions only, will the transfer "THEN" take place.

EXERCISES 2.4

1. In our school intramural wrestling league, four weight classes were established. This program was written to determine the weight class for the wrestlers. Trace the program by using a table of STORED VALUES and the properly formatted OUTPUT.

```
5 PRINT "WEIGHT ROSTER"
10 READ N$,W
20 DATA "J. SMITH", 130, "A. FORBES", 174, "S. PLOD", 87
30 DATA "N. JONES", 210, "S. REED", 101, "F. COOK", 138
40 DATA "R. ALLEN", 85, "W. TRASK", 122, "D. WHEEL", 160
45 DATA "STOP", 0
48 IF N$ = "STOP" THEN 170
50 PRINT N$,
60 IF W<100 THEN 110
70 IF W<125 THEN 130
80 IF W < 150 THEN 150
90 PRINT "HEAVYWEIGHT"
100 GO TO 10
110 PRINT "FLYWEIGHT"
120 GO TO 10
130 PRINT "LIGHTWEIGHT"
140 GO TO 10
150 PRINT "WELTERWEIGHT"
160 GO TO 10
170 END
```

2. Before learning the skills of solving equations, students think that they can guess the solution of any equation. Write a program to allow the user to guess the correct solution of this equation.

$$X/5 = 17 - (3X/2)$$

If he/she answers correctly, congratulations are in order. If not, allow for more guesses.

3. We discussed "ABSOLUTE VALUE" earlier. We are fortunate that the Apple can find the absolute value of a number, using the ABS function. If there were no ABS function, could we still find the absolute value? Write a program to read in several values and find their absolute value. Look at these examples to stimulate your memory about absolute value.

$$|5| = 5$$
$$|-5| = 5$$

4. Both of the following programs have flaws concerning branching. Through tracing, find and correct all errors.

```
        A.) 10 PRINT "WHAT IS CAPITAL OF MAINE?"
20 INPUT MS
30 IF MS = "AUGUSTA" THEN 50
40 PRINT
50 PRINT "CORRECT, THE SOLUTION IS MS"
```

B.)

```
10 PRINT "NEGATIVE, POSITIVE, OR ZERO"
20 READ N
25 DATA -3,6,0,9,-4
30 IF N = 0 THEN 60
40 IF N < 0 THEN 80
50 IF N > 0 THEN 100
60 PRINT N, NEGATIVE
70 GO TO 20
80 PRINT N, POSITIVE
90 GO TO 20
100 PRINT N, ZERO
110 GO TO 20
```

How can we shorten this last program?

5. Speaking of superhuman powers, Captain Apple can even tell if two words are alphabetically in order. He uses the symbols, < and >, for string alphabetizing. To him "APPLE" < "COMPUTER" because Apple comes before computers in the dictionary. Write a program to input two words and print them out alphabetically.

The output should list the two words and then alphabetize them:

```
      WORDS             ALPHABETICAL

VARIABLE,STRING     STRING,VARIABLE
  LET,PRINT            LET,PRINT
```

SECTION 2.5 WE CAN SOLVE ALL THOSE ERRORS, IF...

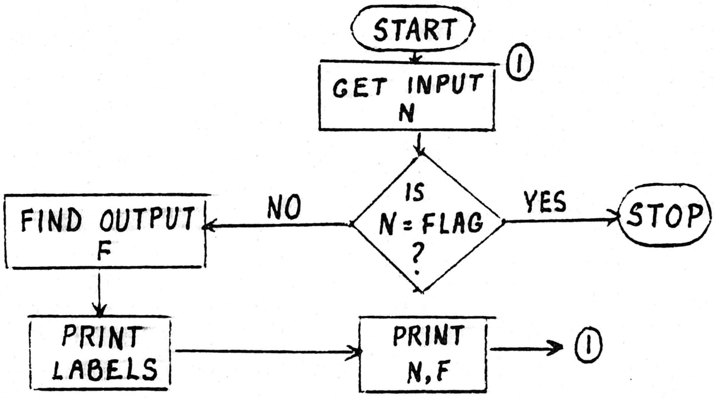
Rather than look at a new problem, we would like to return to an old one. At this point, we still have an error each time we use INPUT or READ-DATA along with the GO TO statement. With the INPUT statement, we have an endless program which can only be stopped with CRTL-C at the keyboard. With the READ statement, the program ends with an OUT OF DATA ERROR. But with our conditional transfer, we can solve these problems. No more Perpetual Motion Machines!

Our project problem is to have the Apple perform as a function machine. Allow for inputting and prevent the endless program error in this project.

Rather than rewrite the program, we have the program from Section 2.2:

```
10 INPUT "INPUT FUNCTION VALUE: "; N
20 LET F = 2 * N + 1
30 PRINT "FUNCTION: F(N) = 2 * N + 1"
40 PRINT "INPUT", "OUTPUT"
50 PRINT
60 PRINT N, F
70 PRINT
80 GO TO 10
```

Now what can we program into this as a stopping mechanism? The stopping mechanism revolves around what is called an input "flag". The programmer selects a value (usually somehow inappropriate for the actual input values) which will serve as an indicator or "flag" that the operator is finished. Each time an input is entered, the program will check to see if the input matches that special indicator flag. If it does not, the program proceeds as usual. If it does, however, then we wish to jump over the entire program and stop. The flowchart looks like this:



It is important that the keyboard operator know what the flag is. And in fact, it is a good policy to remind him/her of it each time the program requests input. This can be done using the special print capability of the INPUT statement. If 999 is chosen as the input flag our INPUT statement might look like:

```
10 INPUT "INPUT A FUNCTION VALUE (TYPE 999 TO STOP): ";N
```

The operator will have no question as to how to terminate the program.

But how will the flag statement look? It is a decision and hence is translated as an IF-THEN statement. The IF sentence will compare the inputted value to the flag, 999. Remember if the answer is NO, then the rest of the program will sequentially follow the conditional transfer.

```
IF N = 999 THEN
```

The program would now appear as:

```
10 INPUT "INPUT A FUNCTION VALUE (TYPE 999 TO STOP): ";N
15 IF N = 999 THEN
20 F = 2 * N + 1
30 PRINT "FUNCTION: F (N) = 2 * N + 1
40 PRINT "INPUT", "OUTPUT"
50 PRINT
60 PRINT N, F
70 PRINT
80 GO TO 10
```

But we have the transfer to deal with. The YES path consists of a termination of the program. Here we need a statement that will stop it. In Applesoft the END statement does this. It looks like:

```
90 END
```

The END statement must be the final statement in the program and always have the highest line number. This statement could have been used at the end of every program. It is only required when we need to branch to the end of the program. We can complete the conditional transfer as:

```
15 IF N = 999 THEN 90
```

Our program is now complete.

Before moving to our READ-DATA problem, let's take time to consider some options in the INPUT statement to flag the end of the program. It certainly is possible to direct the user in a prior PRINT statement. This might be necessary if there are long elaborate directions on what values need to be inputted. Special considerations should be used if the input makes assignments for more than one variable: INPUT X, Y. Here you need to indicate which variable serves as an indicator. And remember, before the conditional transfer is executed, both values will need to be inputted, even if only one acts as a flag.

```
5 PRINT "TYPE IN TWO NUMBERS, THE FIRST SHOULD"
6 PRINT "BE THE LARGER.  IF YOU WISH TO STOP"
7 PRINT "INPUT ZERO FOR THE FIRST AND"
8 PRINT "ANY OTHER SECOND NUMBER."
10 INPUT A, B
12 IF A = 0 THEN 999 (999 END)
```

Another common way of handling the termination of an INPUT program is to use a separate INPUT statement for the flag. In this way, the keyboard user must first indicate whether he/she wishes to use the program. Only if he/she responds affirmatively will the program ask for input values. Within this flag input, strings can be employed easily.

```
5 INPUT "DO YOU WISH TO USE THE FUNCTION MACHINE
        (YES/NO)? "; AS
7 IF A$ = "NO" THEN 999
10 INPUT "INPUT FUNCTION VALUE: "; N
```

In this style, each time the program starts over, the user will have to answer the question, "D0 YOU WANT TO USE THE FUNCTION MACHINE?", before continuing. In the first format, he/she simply returns to the program by not typing the flag. It is a matter of programmer choice -- you have control.

Before leaving the flag idea, we want to consider the OUT OF DATA ERROR. Here is the program we started with, except that we have replaced INPUT with READ-DATA.

```
10 READ N
15 DATA 15, 25, -3
20 LET F = 2 * N + 1
30 PRINT "FUNCTION: F(N) = 2 * N + 1
40 PRINT "INPUT", "OUTPUT"
50 PRINT
60 PRINT N, F
70 PRINT
80 GO TO 10
```

The technique for stopping is quite simple. We use a piece of data called "DUMMY DATA" to serve as the flag. It is included to act as the indicator. The new data statement is:

```
15 DATA 15, 25, -3, 999
```

Here 999 serves as the indicator. Our next statement is the flag conditional transfer:

```
18 IF N = 999 THEN
```

And again the program will need an END statement. It looks like this finally:

```
10 READ N
15 DATA 15, 25, -3, 999
18 IF N = 999 THEN 90
20 LET F = 2 * N + 1
30 PRINT "FUNCTION MACHINE: F(N) = 2 * N + 1
40 PRINT
50 PRINT "INPUT", "OUTPUT"
60 PRINT N, F
70 PRINT
80 GO TO 10
90 END
```

As with the INPUT statement, be careful when reading in more than one variable at a time. Be sure to have as many pieces of dummy data as variable names in the READ statement.

```
120 READ A, B, C$
130 DATA 1, 6, "TOY", 2, 17, "BALL", 999, 999, "NOTHING"
140 IF A = 999 THEN
```

In this DATA statement, the last three pieces all are dummy data. We only need to test A's value. Without all three values there, we will still have an OUT OF DATA ERROR.

Previously we have used the IF-THEN statement to branch our program to various paths. We can use the IF-THEN statement to test for a flag or indicator. Thereby we can place into our program the means to terminate it.

EXERCISES 2.5

1. The following program finds the total amount of change a person is carrying. In the program the variables represent the following: H -- half dollars, Q -- quarters, D -- dimes, N -- nickels, P -- pennies. You are to rewrite the program so that it includes:

A.) An explanation of the program to the user
B.) Input direction
C.) An appropriate stopping mechanism
D.) Meaningful output

```
10 INPUT H,Q,D,N,P
20 LET C = H * .50 + Q * .25 + D * .10 + N * .05 + P * .01
30 PRINT C
40 GO TO 10
```

2. Both of the following programs have an error involving the use of the flag. Through tracing, find and correct the errors.

A.)

```
10 PRINT "EXAMPLE 1"
20 INPUT "INPUT WORD (TYPE STOP TO END):";"NS
30 PRINT MID$(N$,2,LEN(A$)-2)
40 IF N$ = "STOP" THEN 60
50 GO TO 20
60 END
```

B.)

```
10 PRINT "EXAMPLE 2"
20 READ A$,B$
30 DATA "HELLO", "THERE", "COME", "NOW", "STOP"
40 IF A$ = "STOP" THEN 70
50 PRINT B$; SPC(2); AS
60 GO TO 10
70 END
```

3. Often programs are written so that the user must respond in the affirmative or negative -- YES or NO. Unfortunately, to some of us, YES can be substitued for by YEP, YA, YEAH, YESSIR, etc. Write the necessary question to allow any of these to be acceptable answers.

```
10 INPUT "YES OR NO? ";AS
```

input value may not be what is asked for. Therefore, built into some programs is a check to see if the input is acceptable. We call this BULLETPROOFING. The program below finds the square of a positive number. If you remember, no negative number has a square root, and having a negative argument in the SQR function will return an ERROR. Create the necessary statements to prevent a negative value input. If a negative number is inputted, you might remind the user of the directions and ask for another input.

5. The secretary pool is unfortunately overstaffed. Some pink slips need to be handed our. Any secretary who cannot type eighty words per minute must be fired. Write a program to check the skill level of the four secretaries. For any nonqualified secretaries, output a "NOTICE".

SECRETARY	WORDS/MINUTE
HAIG	93
WATT	24
WEINBERGER	75
SCHULTZ	86

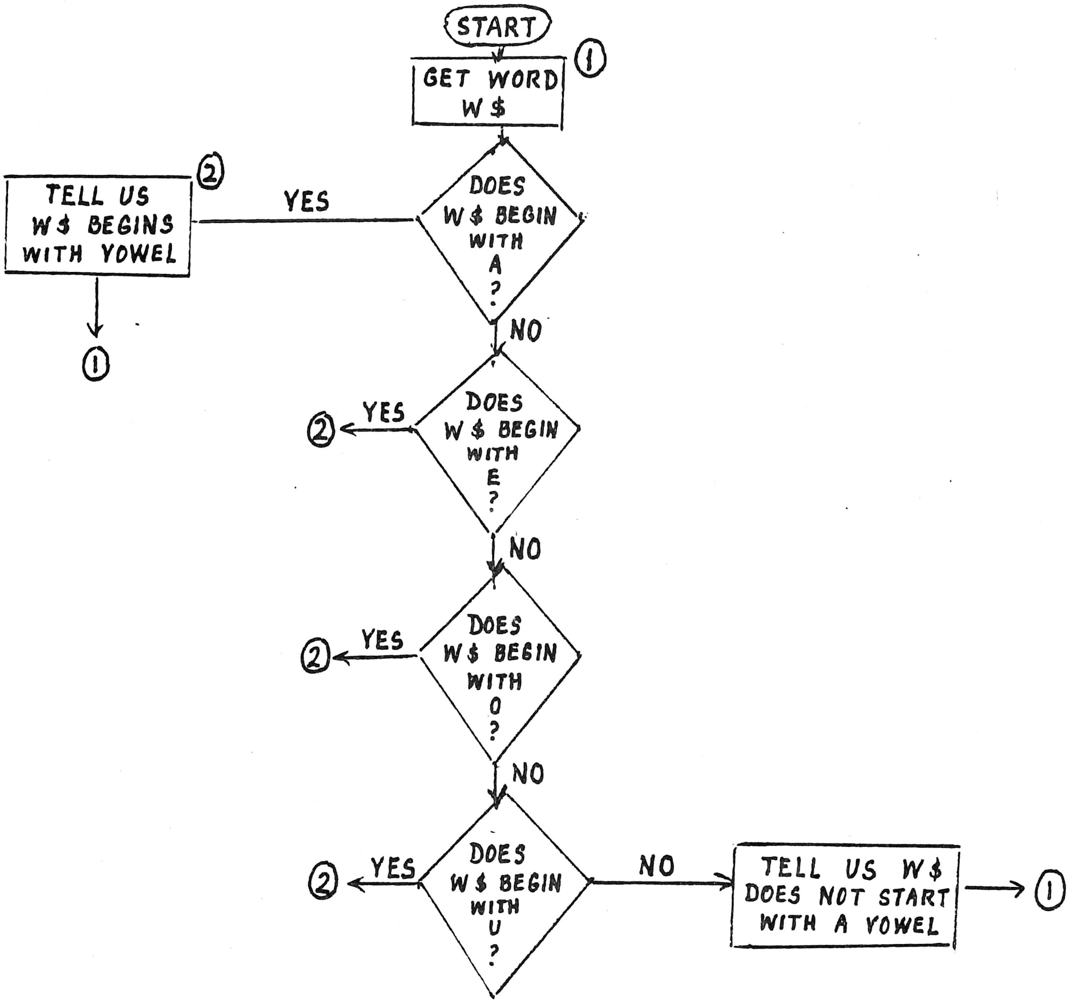
Be sure to avoid the OUT OF DATA ERROR.

SECTION 2.6 NO IF'S AND'S OR BUT'S

Conjunctions -- and, or, but, if, for -- play an essential role in the English language. They give us the option of combining sentences to form compound sentences. In mathematics, the conjunctions "and" and "or" are an important part of set theory. Our next project will explore the role of conjunctions in Applesoft.

The English language is unusual in that five letters out of the entire alphabet have a superior role in the make-up of words. It is estimated that 43% of written language is made up of the five vowels. And yet only 23% of our words begin with vowels. Write a program that will input a word and tell us if it begins with a vowel.

Remembering that words beginning with "a", "e", "i", "o", or "u" will satisfy our requirement, let's attempt to flowchart the problem.


START
GET WORD W$
①
DOES W$ BEGIN WITH A ?
YES
TELL US W$ BEGINS WITH VOWEL
②
①
NO
DOES W$ BEGIN WITH E ?
② YES
NO
DOES W$ BEGIN WITH O ?
② YES
NO
DOES W$ BEGIN WITH U ?
② YES
NO
TELL US W$ DOES NOT START WITH A VOWEL
①

The first diamond box indicates a flag to terminate the program. We could use any of our techniques from the previous section to accomplish this. What follows is a lengthy series of questions, each asking if the first letter is one of the vowels. All of the YES paths transfer to the PRINT statement -- "W$ BEGINS WITH A VOWEL". The NO path eventually leads to a PRINT statement with the other conclusion. Take some time and see if you are able to translate this into an Applesoft program. Do that before you look at our program.

```
10 INPUT "INPUT ANY WORD (TYPE STOP TO END): "W$
15 IF W$ = "STOP" THEN 110
20 IF LEFT$ (W$, 1) = "A" THEN 90
30 IF LEFT$ (W$, 1) = "E" THEN 90
40 IF LEFT$ (W$, 1) = "I" THEN 90
50 IF LEFT$ (W$, 1) = "0" THEN 90
60 IF LEFT$ (W$, 1) = "U" THEN 90
70 PRINT W$; " DOES NOT BEGIN WITH A VOWEL."
80 GO TO 10
90 PRINT W$; " DOES BEGIN WITH A VOWEL."
100 GO TO 10
110 END
```

In comparing your program with ours, did you use LEFT$ or MID$? How did you use the flag to terminate? Did you forget the final GO TO statement or the END statement? Are all your transfer line numbers correct?

This program is certainly not an efficient model, but we can use conjunctions to eliminate all those decisions. Within the conditional part of the IF-THEN statement it is possible to have a compound sentence using the conjunction "AND" or "OR".

A quick review of the significance of these conjunctions follows. In determining the truth value of a compound sentence, you must know the truth value of each part and you must know the conjunction being used. A compound sentence using the conjunction "AND" is true if and only if both of the sentence parts are true. Only if X = 10 is true and Y = 10 is true will the conjunction X = 10 AND Y = 10 be true. The conjunction "OR" works differently. A compound sentence using "OR" is true if either of the sentence parts is true. If X = 10 is true and Y = 10 is false, the conjunction X = 10 OR Y = 10 is still true.

In our problem, if the word starts with any of the five vowels, then it begins with a vowel. Only one of the conditions need to be true to get a successful word. Therefore the conjunction "OR" is needed. The conditional transfer then becomes:

```
20 IF LEFT$ (W$, 1) = "A" OR LEFT$ (W$, 1) = "E" OR
   LEFT$ (W$, 1) = "I" OR LEFT$ (W$, 1) = "O" OR
   LEFT$ (W$, 1) = "U" THEN ____
```

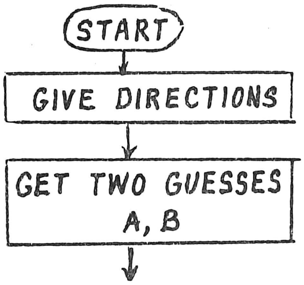
Line 40 replaces lines 20-60. We are in effect asking the Apple to make several decisions and then make an ultimate decision depending on the conjunction. The final program becomes:

```
10 INPUT "ANY WORD (TYPE STOP TO END): "; W$
20 IF W$ = "STOP" THEN 110
30 IF LEFT$ (W$, 1) = "A" OR
   LEFT$ (W$, 1) = "E" OR
   LEFT$ (W$, 1) = "I" OR LEFT$ (W$, 1) = "O"
   OR LEFT$ (W$, 1) = "U" THEN 90
70 PRINT W$;" DOES NOT BEGIN WITH A VOWEL."
80 GO TO 10
90 PRINT W$;" DOES BEGIN WITH A VOWEL."
100 GO TO 10
110 END
```

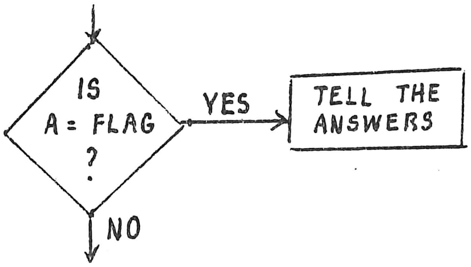
In order to get a feel for using the "AND" conjunction and as a review of our work with conditional transfer, we would like to work on one more programming problem. Together we will construct the flowchart. Translating it into Applesoft will be left as an exercise.

Write a program to have the keyboard player guess two numbers whose sum is 22 and whose product is 105. For the inputted values, check the conditions and tell the player whether he/she is correct or not. Give the player as many opportunities as he/she wishes, but allow for possible frustration. If the player gives up, give him/her the correct answers.

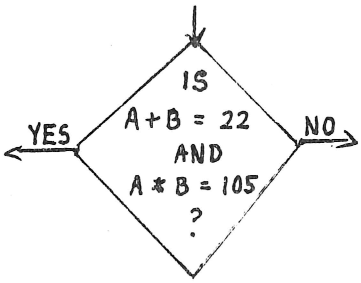
We start our flowchart with directions of the game and ask for the two guess values.



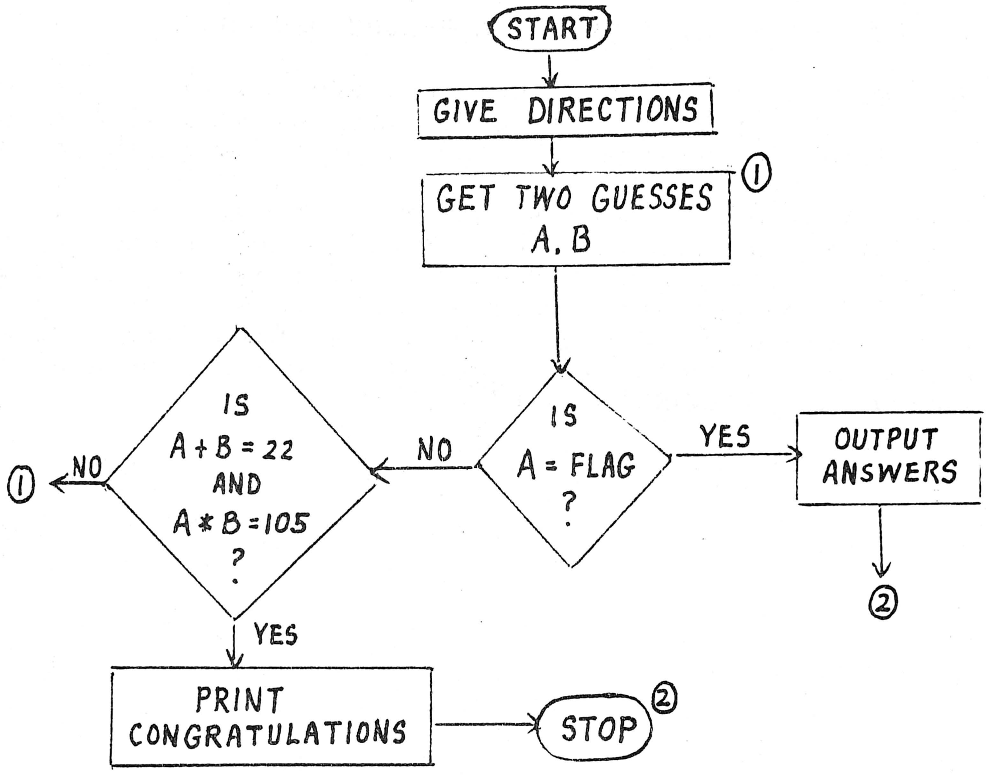
Within these two statements, we should have included the flag value directions. The flag conditional transfer should follow these.



The YES path is an indicator that the player has yelled "UNCLE", so we want to relieve his frustration by outputting the two numbers. The NO path continues the action into the real body of the program. We are ready to test his guesses. Here we can use the "AND" conjunction. Remember both conditions need to be met -- that's the conjunction "AND".



The NO path means that he/she has not successfully found the numbers. Our program should say so. Be encouraging, but send the player back for other inputs. The YES path is another story -- SUCCESS. Use your creative print formatting to congratulate the player. Our final flowchart looks like this:


START
GIVE DIRECTIONS
GET TWO GUESSES A, B
①
IS A = FLAG ?
YES
OUTPUT ANSWERS
②
NO
IS A + B = 22 AND A * B = 105 ?
NO
①
YES
PRINT CONGRATULATIONS
STOP
②

EXERCISES 2.6

1. For each conjunction below, list five integers which satisfy the conditions.
A.) X = 10 OR X > 3
B.) Y < -3 AND Y > -20
C.) X < -3 OR X > 3
D.) (X < 10 AND X > 6) OR (X > 4 AND X < 15)

2. Write the Applesoft IF-THEN statement for each set of conditions.
A.) X$ CAN BE HUMPTY OR DUMPTY
B.) B AND Z MUST BE GREATER THAN ZERO
C.) A = 16 AND Q = 14
D.) THE SUM OF A AND B IS GREATER THAN OR EQUAL TO 10 AND THEIR DIFFERENCE IS LESS THAN OR EQUAL TO 10

3. You are conducting a survey of girls who are doing well in school. You've had every student fill out a data card containing name, sex, and grade average. For example:

```
10 DATA "JOHN KEMENY", "M", 86
11 DATA "GRACE HOOPER", "F", 92
```

Write a program to read all of the data, but print out a list of the names of all girls who also have an average of 85 or better.

4. You're nervously waiting to register to vote. Before you register, you must anwser the following questions truthfully.

A.) HOW OLD ARE YOU?
B.) HAVE YOU EVER BEEN CONVICTED OF A FELONY OFFENSE?

A person can vote only if he is 18 or older and has never been convicted of a felony. Write a program to simulate the registrar. Ask the necessary questions, and determine whether the applicant is accepted or not.

5. The school initiated a weekly lottery as a fund raiser. For a price, each student picks a four letter word. At the end of the week, the principal randomly selects a "WINNER'S" word from the dictionary. If the first letter of the winner's word matches the student's, or if the last letters match, he/she wins a free lunch. If both first letters and last letters match, the student wins $20.00. Write a program to check the student's word with a "WINNER'S" word -- YAK.

A LITTLE BIT MORE SECTION 2

PLACING IT WHERE YOU WANT IT

We have used the PRINT statement in our programming many times. Each time a program is executed the output appears after the program listing. Each output occurs line under line. It is possible within the program to control where on the entire screen the output will be seen. The statement commands for this are HOME, HTAB, and VTAB.

The HOME command is used as a cleaning tool in programming. Upon its execution the screen is erased of all characters, and the cursor is moved to the top left hand corner of the screen. As the beginning statement of a program, it will force the output to occur at the top of a clear screen. The following program indicates this use:

```
10 HOME
20 READ A$,B$,C$,D$
30 "THIS", "IS", "HOME", "COMMAND"
40 PRINT A$
50 PRINT B$
60 PRINT C$
70 PRINT D$
```

Running this program results in the four words, one under the other, outputted at the top left hand corner of the screen. The HOME statement can add elegance to an elaborately tabled output.

The HTAB and VTAB statements also control the location of the cursor for outputting. They will allow us to place output at any spot on the screen. They override the placement of line after line output.

HTAB stands for horizontal tab; VTAB for vertical tab. On the text screen there are forty spaces across the screen and twenty four lines down. VTAB 1 is the top row of the screen, VTAB 24 is at the very bottom. HTAB 1 is the far left of the screen, HTAB 40 is at the far right. To print the first letter of our output "HELLO" in the middle of the screen, we would move the cursor over twenty spaces, HTAB 20, and down twelve spaces, VTAB 12. Thus the program would appear as follows:

```
10 HTAB 20
20 VTAB 12
30 PRINT "HELLO"
```

The program places the word "HELLO" in the middle of the screen. It disregards any other text on the screen. In fact, it would simply write over any other text at that location. A HOME statement prior to using these two commands will clear the output: 5 HOME. Adding this to the program and running it, you will see the "HELLO" in the middle of the cleared screen.

The following program reads in a horizontal and vertical position and a word. Each is outputted at the new location.

```
10 HOME
20 READ A,B,C$
30 DATA 4,2,"TOP LEFT",4,30,"TOP RIGHT"
32 DATA 12,20,"MIDDLE"
34 DATA 24,2,"BOTTOM LEFT"
36 DATA 23,28,"BOTTOM RIGHT"
40 VTAB A
50 HTAB B
60 PRINT C$
70 GO TO 20
```

Here is the output as seen on the screen.

```
TOP LEFT                      TOP RIGHT







                    MIDDLE








BOTTOM LEFT                   BOTTOM RIGHT
```

The difference between HTAB (and VTAB) and TAB function should be discussed. HTAB and VTAB cannot be part of a PRINT statement. The TAB function controls the horizontal position only within a single line of output, regardless of its location on the screen. It works in conjunction with the normal line under line format.

The three statement commands HOME, HTAB, and VTAB can be combined to control the screen formatting. Try experimenting. The results can certainly be very rewarding.

REVIEW EXERCISES

CHAPTER TWO

1. Trace the following program completely with a table of STORED VALUES and properly formatted OUTPUT.

```
10 READ L, A$
20 DATA 1, "THE OPTIMIST FELL 10 STORIES"
22 DATA 2, "AT EACH WINDOW BAR"
23 DATA 3, "HE SHOUTED TO HIS FRIENDS"
24 DATA 4, "'ALL RIGHT SO FAR'"
27 DATA 999, "STOP"
30 IF L = 999 THEN 80
40 IF L = 1 OR L = 3 OR L = 5 THEN 60
50 PRINT TAB(10);
60 PRINT A$
65 PRINT
70 GO TO 10
80 END
```

2. Write a program to analyze some data. Have each member of your class measured. Measure the total height from floor to the top of their heads and measure the waist height from floor to belly button. For each, the program should output both measurements and the ratio of the two. (Did you find anything extraordinary?)

3. Translate each of the following statements into Applesoft.
A.) Transfer to line 10
B.) Read in the value for N and A$
C.) Stop the program
D.) Transfer to line 10 when X is negative
E.) When both X and Y are 10, transfer to 100
F.) If both words, W$, X$, start with A, then go to line 200
G.) The data for X is 2,9,14,16
H.) Go to the end of the program if the dummy data for X is 999

4. Create a multiplication estimating drill. Take two numbers and ask the user to make an estimate of the product of the numbers. If the guess is within 150, above or below, the user should be congratulated. Otherwise report the actual product and how far away the estimate was. Construct a flowchart to create this drill.

5. Prepare a form bill to be sent to your customers. Within the program allow for input of all pertinant information -- name of customer, address, number of units purchased, unit price, and total price. Be creative.

CHAPTER 3

AGAIN AND AGAIN AND AGAIN

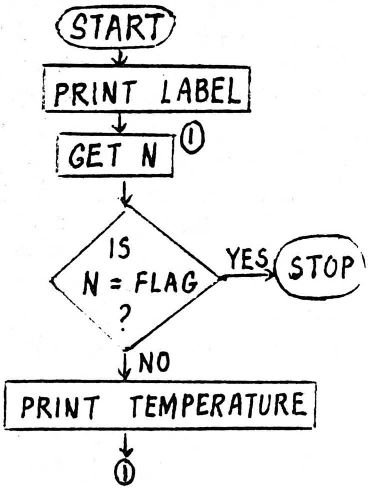
Section 3.1 Super Dooper Looper
Section 3.2 Another Way Around
At The Keyboard: Editing Mistakes Again
Section 3.3 Computer Makes Little Things Count
Section 3.4 Accumulating Loops
Section 3.5 Completely Looped
A Little Bit More: Spicing Up The Print Statement
Review Exercises

SECTION 3.1 SUPER DOOPER LOOPER

One characteristic of the computer is its ability to handle repetitive tasks quickly. Although we have seen this with the unconditional transfer, our next work will concentrate on solving problems with lots of repetition. In our program thus far, we have provided the Apple with all the necessary data. Often the amount of data is so large that it becomes cumbersome. Consider the following problem:

Rubbing their legs together? Is that what makes that chirping noise from the cricket? The cricket's chirp is more than just a territorial claim, it is a thermometer. That is right, you can predict the outside temperature by counting the number of chirps. Here's how to do it. Count the number of chirps for a 15 second period; add 5; and multiply the sum by 5/9. The product is the Celsius temperature! Write a program which will find the Celsius temperature for a 15 second song of from 1 to 20 chirps.

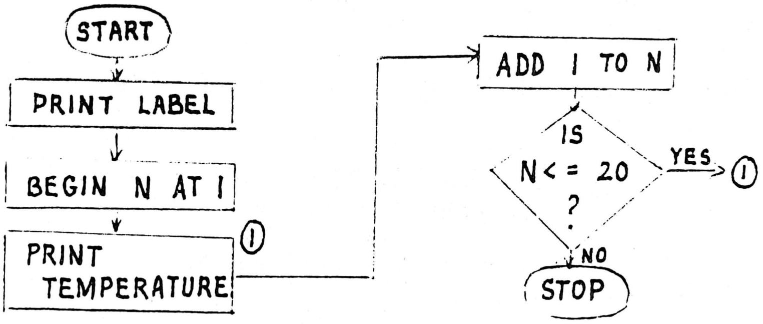
At this point, our ability to solve this problem is confined to the use of READ or INPUT. Here is our flowchart and program.



```
5 PRINT "CHIRPS", "TEMPERATURE"
10 READ N
20 DATA 1,2,3,4,5,6,7,8,9,10,11
25 DATA 12,13,14,15,16,17,18
27 DATA 19,20,999
30 IF N = 999 THEN 70
40 PRINT
50 PRINT N, (N + 5) * 5/9
60 GO TO 10
70 END
```

This program is straightforward, although there is a great deal of data to be input. But imagine the DATA statement if we were required to print out the temperatures for 1 to 50 chirps or 1 to 250 chirps. There must be an easier way. Indeed, the BASIC language is designed for dealing with problems with simplicity and speed. So instead of supplying this data, we will instruct the Apple to generate this information. The technique is referred to as LOOPING.

Study the flowchart below:



The body of the program which outputs the temperatures is the same. But the means of creating the numbers from 1 to 20 is quite different. These instructions are the essential parts of a loop.

The looping process starts by assigning a beginning value to the variable. Here the instruction is "BEGIN N AT 1". This is called the initializtion of the variable. A LET statement is used for this initialization. The loop can commence at any desired value. LET N = 10 would begin our data at 10.

Once this initial value has been used for finding and outputting the temperature, the next crucial part takes place. The value of the variable N must be increased or incremented. We wish to update N's value to 2 so that the temperature can be calculated again. The flowchart instruction to do this is "ADD 1 TO N". This increment step demonstrates a clever use of our storage system to give N a new value. The Applesoft statement looks like:

```
LET N = N + 1
```

The value of N + 1 (one more than what is presently stored in N) will be the new value stored in N. In our program the first time this instruction is encountered, 3 will be the new N value stored.

Just as we can initialize the variable at any value, we can also increment by any value. The statement: LET N = N + 2 would generate the numbers 1, 3, 5, 7, etc. The statement: LET N = N - 1 would decrease the N value by one each time it occurs.

The final parts of a loop determine whether the repetition continues or not. These parts are a conditional test and a transfer. The test is an IF-THEN statement which will test N's value against a terminating value that we determine. In our program, we want the loop to continue as long as N's value is less than or equal to 20. The test value is 20. Changing this value determines the number of repetitions. If our test was 250, our program would continue to calculate temperatures for 1 to 250 chirps. So as you can see, a slight change in a test can make a dramatic difference.

Depending on how the IF-THEN question is phrased, the transfer statement might take two forms. It may be contained in the transfer portion of the IF-THEN. Or it may take the form of an unconditional transfer -- GO TO -- which follows the IF-THEN. Both forms are shown below.

```
50 IF N < = 20 THEN                      50 IF N > 20 THEN 70
   (BACK TO THE LOOP                     60 GO TO (BACK TO THE LOOP
   BEGINNING)                               BEGINNING)
60 END                                   70 END
```

The final complete program then appears:

```
10 PRINT "CHIRPS", "TEMPERATURE"
20 LET N = 1
30 PRINT N, (N + 5) * 5/9
40 LET N = N + 1
50 IF N < = 20 THEN 30
60 END
```

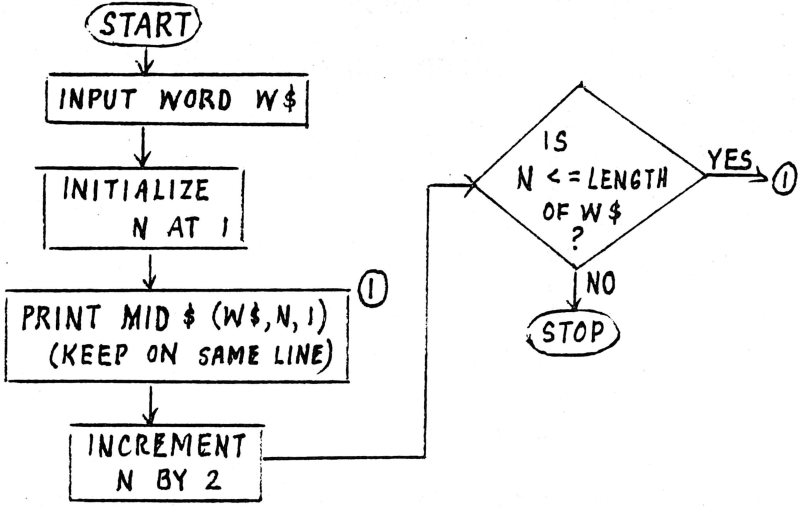
For each looping program all four parts must be present:

1.) INITIALIZATION (LET N = 1)
2.) INCREMENT (LET N = N + 1
3.) TEST (IF N < = 20)
4.) TRANSFER (...THEN 30)

Another example demonstrating these four parts of a loop is shown below.

A message from M is received by 007 -- "EEY AET ARVS A AROT 80." Bond interprets the code and snaps into action. Your mission, should you decide to accept it, is to write a program to produce a secret coded word. Allow the user to input any word and print out a new word leaving out every other letter. Therefore an input of APPLESOFT would yield a coded output of APEOF.

A brief discussion before we flowchart the solution will help us get organized. After getting the inputted word, we want to start our loop. What is being looped? We need to look at one letter at a time, so what is being looped is the argument of MID$. Since we want every other letter, the increment will be 2. What about a test? Well, since we are looking at each letter, we need to loop up to the number of letters in the string. What function relates to this? A flowchart should help collect these ideas.



Try translating this into a BASIC program. Think about what the output should look like. But concentrate on the four parts of the loop.
The Agency is counting on your unique skills to complete this mission.

EXERCISES 3.1

1. The programs below are looping programs used to count. Fill in the statements to complete the looping process.

A.)

```
10 PRINT "COUNTING BY 12'S FROM 0 TO 100
20 LET X =
30 PRINT X
40
50 IF X =
60 END
```

B.)

```
100 PRINT "COUNTING DOWN BY 2'S FROM 20 TO -20
200
300 PRINT
400
500
600 END
```

2. The Apollo space project put America on the moon. Simulate the final countdown for the Lunar mission.

```
15 SECONDS AND COUNTING

10
 9
 8
 7
 6
 5
 4
 3
 2
 1
BLAST OFF!
```

3. The following program evaluates the algebraic expression (N -1)/(N - 1) - N for the values 1 to 5. Trace the program completely.

```
10 LET X = 1
20 PRINT X, (X ^ 2 - 1)/(X - 1) - X
30 LET X = C + 1
40 IF X <= 5 THEN 20
50 END
```

4. As you can see from our examples, word play is a favorite pasttime. Write a program to input a word and print it out backwards. Consider using the LEN function in your initialization.

5. America has some promising long distance women runners for the upcoming Olympics. But Americans still have trouble converting those metric long distances to mile measure. Write a program to generate a table of conversion METERS TO MILES. Step up by 100's from 100 meters to 10,000 meters.
(HINT -- 1.1 yards to a meter; 1760 yards to a mile. ENOUGH SAID.)

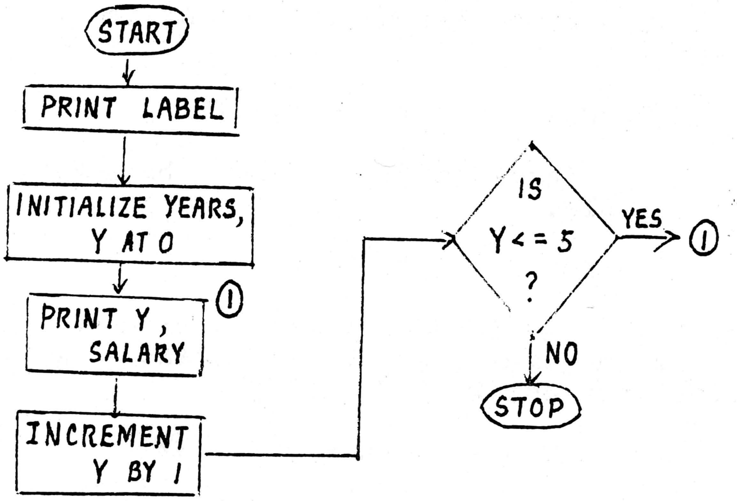
SECTION 3.2 ANOTHER WAY AROUND

We have seen how a loop enables us to generate a large sequence of numbers, where there is a pattern in the sequence. Every loop has four essential parts, remember? -- initialization, increment, test, and transfer. But there is alternative language in Applesoft to execute a loop. We'll explore this new vocabulary in this project.

The Major League Baseball Players Association came to the bargaining table with a new contract demand for minimum salaries. They suggested that the minimum player salary be $20,000 plus $5,000 for every year of experience. Write a program that will give the minimum salary of a player from his rookie year to his fifth year. The output should look like this:

```
YEARS EXPERIENCE     MINIMUM SALARY
       0                 $20000
       1                 $25000
```

With your experience in constructing loops, you should be able to construct a flowchart for this problem. Study ours only after doing some work on your own!



Although the flowchart has not changed for this new form of loop, the Applesoft statement combines the four parts in a different way. All four looping parts are contained in two statements -- FOR and NEXT statements.

The FOR statement contains the initialization and is located in the flowchart at the instruction "START X AT 0". However, within the FOR statement, the increment value and test flag value are also found. The general statement might look like:

```
FOR X = 0 TO 10 STEP 1
```

The variable is assigned an initial value. The value after "TO" is the ending (flag) value. The word STEP is used to indicate increments, here by one's. If we wished to loop by two's from 20 to 40, the statement would look like:

```
FOR X = 20 TO 40 STEP 2
```

The loop statement for our countdown program would be:

```
FOR X = 10 TO 0 STEP - 1
```

The NEXT statement replaces the transfer box in our flowchart. Its form is:

```
NEXT X
```

When the FOR statement is executed the first time, the value 0 is stored in X. Then when the NEXT statement is encountered, the Apple increments X by the STEP value and tests to see if that value is greater than the flag. If it is not, the program transfers back to the line following the FOR statement. If the incremented value exceeds the test, the program proceeds to the line following the NEXT statement. Whew!

Let's translate the flowchart to Applesoft, one statement at a time. We begin by outputting the column headers -- "YEARS EXPERIENCE" and "MINIMUM SALARIES". These titles need to be outputted prior to the start of the loop. We do not want the titles printed each time the loop is repeated. However, if we wanted another formatting -- perhaps sentence form -- then the output would certainly occur inside the loop. Line 10 becomes 10 PRINT "YEARS EXPERIENCE"; TAB(22); "MINIMUM SALARY".

Following this is the initialization of the loop, but we will use the new FOR statement.

```
20 FOR Y = 0 TO 5 STEP 1
```

But in fact, if the increment is by one, the STEP portion may be omitte Without STEP, the Apple...

```
20 FOR Y = 0 TO 5
```

...understands that the increment is 1. With our loop begun, we now need to calculate the minimum salary. Adding the starting salary to the experience factor increase will do the job.

```
30 LET S = 20000 + Y * 5000
```

Our output should be in column form to match with our column title.

```
40 PRINT TAB(9); Y; TAB(26); "$"; S
```

In our flowchart, the subsequent statements constitititute the increment, test and transfer. All of these are contained in a single statement -- the NEXT statement.

```
50 NEXT Y
```

The first time NEXT Y is executed the following things occur. First, Y is incremented by one. The new Y -- 2 -- is then tested against the end value 5. Since 2 is not greater than 5, the program is transferred to line 30 (the statement following the FOR statement.) Despite the length of the NEXT statement you can see it performs many duties.

To complete the program we need to finish the NO path with an END statement. The YES path has already been included in the NEXT statement. The completed program is then:

```
10 PRINT "YEARS EXPERIENCE"; TAB(22); "MINIMUM SALARY"
20 FOR Y = 0 TO 5
30 LET S = 20000 + T * 5000
40 PRINT TAB(9); Y; TAB(26); "$"; S
50 NEXT Y
60 END
```

In order to better understand looping and in particular FOR-NEXT loops, we would like to trace this program. But rather than step by step, we will look at the tracing at various points in the program. After going through the loop three times, the trace looks like:

STORAGE VALUES		OUTPUT	
Y	S	YEARS EXPERIENCE	MINIMUM SALARY
0	20000	0	$20000
1	25000	1	$25000
2	30000	2	$30000

At this point in the program, following the output in line 40, the NEXT statement increments and tests the new value. Since 3 passes the test, the program transfers to line 30 again.

Two more times through and the trace now looks like this:

STORAGE VALUES		OUTPUT	
Y	S	YEARS EXPERIENCE	MINIMUM SALARY
0	20000	0	$20000
1	25000	1	$25000
2	30000	2	$30000
3	35000	3	$35000
4	40000	4	$40000

Once again, when the NEXT statement is executed, Y is incremented and tested. Since 5 is not greater than the test value 5, the loop continues. However, the next time through Y is incremented to 6 and tested. Since 6 > 5, it does not pass the test. The condition is not met, and the program (as it does with all IF-THEN statements) proceeds to the subsequent statement. In our program the execution is terminated with the END statement. The tracing at that point looks like:

STORAGE VALUES		OUTPUT	
Y	S	YEARS EXPERIENCE	MINIMUM SALARY
0	20000	0	$20000
1	25000	1	$25000
2	30000	2	$30000
3	35000	3	$35000
4	40000	4	$40000
5	45000	5	$45000

The FOR-NEXT statements can replace any traditional loop, assuming the data is incremented by a constant amount. All the parts of a loop are included, but the way in which they are included is quite different.

As a quick review of this new looping vocabulary, we give you a chance to interpret several short programs. (Bet you can't wait!) Each program uses FOR-NEXT statements. For each, predict the output. Carefully consider initialization, increment and test before you make your conclusions.

```
A.)  10 FOR X = -10 TO 10 STEP -2
     20 PRINT X; SPC(2);
     30 NEXT X
     40 END
```

```
B.)  10 FOR X = 3 TO 12 STEP 4
     20 PRINT X; SPC(2);
     30 NEXT X
     40 END
```

```
C.)  10 FOR X = 18 TO 24
     20 PRINT X; SPC(2);
     30 NEXT X
     40 END
```

```
D.)  10 FOR X = 50 TO 0
     20 PRINT X; SPC(2);
     30 NEXT X
     40 END
```

```
ANSWERS A.) OUTPUT:  -10  -8  -6  -2  0  2  4  6  8  10
        B.) OUTPUT:  3  7  11
        C.) OUTPUT:  18  19  20  21  22  23  24
        D.) OUTPUT:  50 (THE TEST FAILS THE FIRST TIME "NEXT C"
                         IS ENCOUNTERED)
```

EXERCISES 3.2

1. The T.V. weatherman is still giving temperatures in Fahrenheit, so we wrote him a program to help him convert to Celsius. Rewrite the program using a FOR-NEXT loop.

```
5 PRINT "FAH", "CELSIUS"
10 LET F = -30
20 PRINT F, 5/9 * (F-32)
30 LET F = F + 10
40 IF F <= 100 THEN 20
50 END
```

2. The following programs contain looping mistakes. Use the label to help debug the programs.

A.)

```
10 PRINT "MULTIPLES OF 7 UNDER 100"
20 FOR X = 1 TO 100 STEP 7
30 PRINT X
40 NEXT X
```

B.)

```
10 PRINT "99 BOTTLES OF BEER ON WALL"
20 FOR X = 98 TO 1
30 PRINT X; "BOTTLES LEFT ON WALL"
40 NEXT X
```

C.)

```
10 PRINT "HALVES FROM 100 TO 200"
20 FOR X = 100 TO 200 STEP .5
30 PRINT X; SPC(2);
40 GO TO 20
```

3. Write a program to give a special artistic look to your name. Input any name and have the output look like:

```
A
AP
APP
APPL
APPLE
```

(HINT: Use the LEN function in both the test and the PRINT statement)

4. One approach to solving equations is TRIAL-and-ERROR. Although it is very inefficient for you, Apple can attempt it with much more speed. Have the Apple test the values from 0 to 50 to see if any are solutions to the equation: $(x + 2)/3 = (2x - 7)/5$. Print out only those values which are solutions.

5. Rewrite any necessary portions of the above program to modify the boundaries. If there had been no solution found from the values 1 to 50 in the above equation, we would have wanted to check another set of numbers. Or we would like to change the step to include possible rational number solutions. Modify any necessary statements in your program to allow the user to input the loop boundaries and increment. You might make a point of checking his/her input to see if the user is demanding to perform a loop repeated too many times.

AT THE KEYBOARD SECTION 3

EDITING MISTAKE AGAIN

You might think we are over-emphasizing "CORRECTION", but fixing errors is a well known hobby of programmers. In Chapter One we discussed how to make corrections. Retyping the line is a simple procedure, but it may be a time consuming one if the line is complex.

There is another method that does not involve retyping. It does involve using the <u>EDITOR</u> mode of Applesoft. And it does take practice to master. Look on the keyboard for the ESC key (ESCAPE). This is the <u>EDITING KEY</u>. We will see how to use it to correct the following line.

```
20 FOR X - 1 TO 30 STEP-2
```

Again we wish to replace information, but this time the only character that needs correcting is the "-". Follow the procedure outlined below:

A.) Press the escape key once. We are now in the EDITOR mode. This enables the cursor to be moved to any location on the screen.
B.) The four keys that are used to move the cursor are the I, J, K, and M. I moves the cursor up; M is down; K is right; J is left. Press the I key to move the cursor to line 20.
C.) Just when you think you've got it, a small complication must be dealt with. You might think that you should move the cursor over to the mistake by using the J or K keys, make the correction, and press RETURN. But this is only partially correct. Before correcting the mistake you must start at the very beginning of the line, the first digit of the line number. If you look at our example, the cursor is over the 0 in 30. Use the J key to move the cursor to the first location, 3.
D.) We are ready to correct our mistake. The Apple requires us to reread over the entire line as we are correcting. The right arrow key moves the cursor and also rereads what it passes over. When we get to the mistake, correct it by retyping, in our example "=". Then finish reading the line by moving the cursor (using the right arrow key) to the end of the line. Press RETURN and our line is fixed. List it for your confidence in using this mode.

Two points need to be clarified. One question might be: Why not use the K key to read the line, instead of the right arrow key? The answer is simple. The editing keys do not read a line. They just pass over the letters. Only the arrow key will put the line into the computer's memory. The second point is apparent if you have already tried to edit. You may have noticed that you had to press the arrow key twice before it moved. When pressed the first time, it signals the Apple to leave the EDIT mode. The second time the key will begin to read the line.

In summary, the four commands are as follows:

A.) Press ESC once.
B.) Move the cursor to the beginning of the incorrect line using the I, J, K, and M keys.
C.) Use the right arrow key to read the line, correcting your mistakes by retyping.
D.) Press RETURN when you have reached the end of the line.

It is worth repeating this sequence. This process needs practicing. Your first attempts may not be successful. Part of a line might be deleted; spaces may be read in by mistake; the entire line may be misread. Practice does indeed make PERFECT corrections.

For those of you who have a good understanding of the editing features, there is one other mode you may want to investigate -- the INSERT mode. Suppose you typed the line:

```
10 PRINT "THE COW JUMPED OVER THE MOON"
```

But what you wanted to type was:

```
10 PRINT "THE BROWN COW JUMPED OVER THE MOON"
```

You want the word "BROWN" inserted in the line. You have two options available to you. The first is to retype the line the way you wanted it. This may be time consuming and is certainly boring! Try using the editing mode to do your work for you. First enter the EDIT mode by pressing the ESC key. Now move the cursor to the beginning of the line and use the right arrow key to read up to the space between "THE" and "COW". Press the ESC key and then press the I key to move the cursor up one space. The third step is to type the word "BROWN" (you will have to type the "B" twice, once to exit the EDIT mode and once to type the letter). Press the ESC key once more and use the K key to move back over the newly inserted word. Finally use the M key to move down to where you were before starting this extraviganza. Read to the end of the line with the right arrow key, press RETURN and VOILA! If you list the program you will see the insertion you directed.

One warning to prospective editors -- some of the older computers do not have "AUTO-ESCAPE". Therefore to perform any of the procedures described above, the ESC key must be pressed each time the cursor is to be moved. Also the cursor movement keys are different. A moves the cursor up; B to the left; C down; D up.

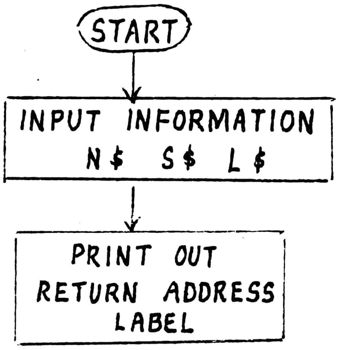
As Confucious say -- May every error be a small one. And delete the others! Good luck fixing those errors.

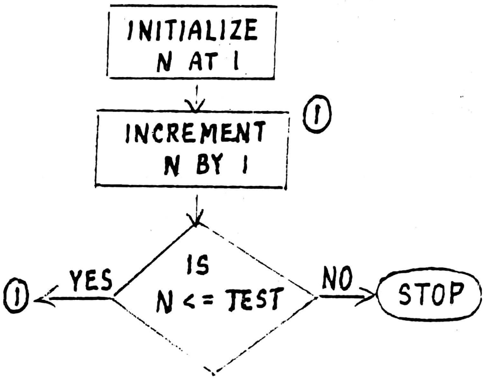
SECTION 3.3 COMPUTER MAKES LITTLE THINGS COUNT

Our looping techniques allow us to perform repeated procedures quite easily. But not only can the Apple do repetitions, it can also count the number of repetitions carried out. We can use our looping procedures in two ways to count. One system will be used when we know how many times a procedure is to be repeated. The second will involve counting how many times a procedure was done. Therefore we will be looking at two programming problems in this counting section.

With your new programming skills, you may as well start using them for some household tasks. Write a program to print out twenty return address labels. Allow the keyboard user to input his name, street address and locality (city, state, and zipcode).

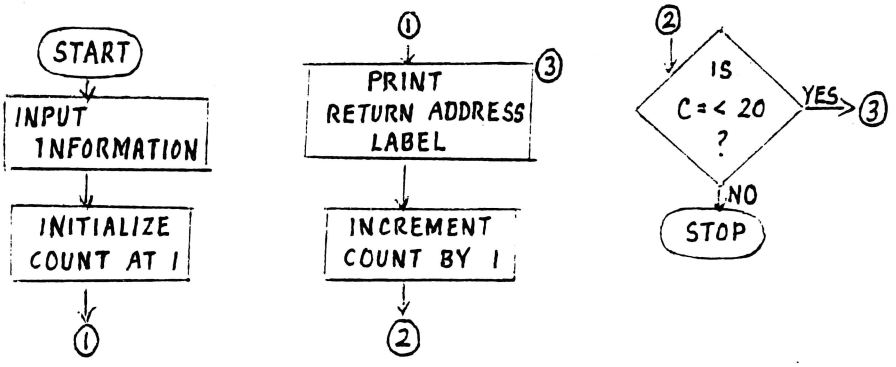
This program is actually two separate programs. One part prints out the label. It will consist of PRINT statements with a particular format. The second part is a counting program.


START
INPUT INFORMATION
N$ S$ L$
PRINT OUT
RETURN ADDRESS
LABEL

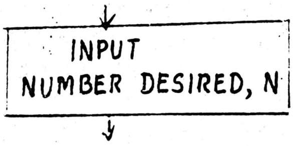

INITIALIZE
N AT 1
INCREMENT
N BY 1
①
IS
N <= TEST
YES
①
NO
STOP

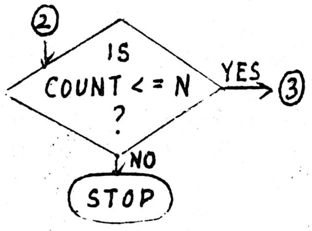
The looping program is not involved in the printout of a label. It does not affect the output at all. It merely will control the number of times that output is performed. This loop is a COUNTING LOOP. It is the ultimate control in the number of repetitions executed.

The merged flowchart for this problem looks like:


START
INPUT
INFORMATION
INITIALIZE
COUNT AT 1
①
①
PRINT
RETURN ADDRESS
LABEL
③
INCREMENT
COUNT BY 1
②
②
IS
C =< 20
?
YES
③
NO
STOP

Before translating the flowchart, we might make one adjustment to show our counting control. Into the program, add the ability to determine the number of labels outputted. The user should be able to input that number. This becomes our new test.





Our program looks like this:

```
10 PRINT: "THIS PROGRAM WILL PRINT OUT YOUR"
20 PRINT "RETURN ADDRESS LABELS.  YOU WILL BE"
30 PRINT "ASKED TO INPUT THE NECESSARY INFORMATION"
40 PRINT "AND THE NUMBER OF LABELS REQUIRED"
50 INPUT "INPUT FULL NAME: " N$
60 INPUT "STREET ADDRESS: " L$
70 INPUT "LOCALITY (CITY, STATE, ZIPCODE): " L$
80 INPUT "NUMBER OF LABELS REQUIRED: " N
90 FOR C = 1 TO N
100 PRINT
110 PRINT
120 PRINT "_______________ ___________________"
125 PRINT
130 PRINT N$
140 PRINT S$
150 PRINT L$
160 NEXT C
170 END
```

The counting loop can be used in any controlled repetition. Whether it be a shape, a form letter, a picture, or a procedure, we can use a counting loop.

An unusual use of a counting loop is in making the Apple pause in its execution. This use is called a DELAY LOOP. Because the Apple works so rapidly and outputs quickly, there may be times when we want to slow down that process. The technique is to place a counting loop prior to the instruction to do a task or in between two tasks.

```
10 PRINT "SLOW"
20 FOR C = 1 TO 500
30 NEXT C
40 PRINT "DOWN"
```

This program begins with the output of the word "SLOW". Then the computer enters the delay loop. This loop simply has the Apple count from 1 to 500. For each time line 30 is encountered it increments C and transfers back to itself. After counting to 500, the program proceeds to output "DOWN". When executed, it appears that there is a pause between "SLOW" and "DOWN". The length of the pause depends on the length of the counting loop. Generally speaking, it takes the Apple about one second to count to 500. Therefore an end value of 3000 would create a pause of six seconds. This delay tactic can be used for emphasis, to increase readability, or to prevent "viewing blitz".

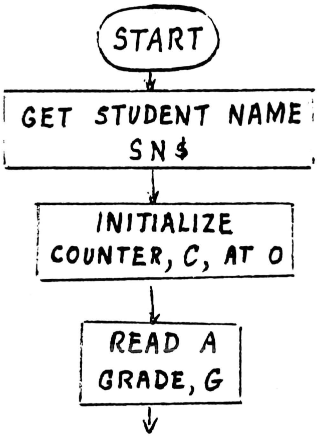
Now to the other counting problem. The second counting technique can be used when we do not know the number of repetitions which will occur. This new strategy will allow the programmer to keep track as each repetition takes place. Here's the problem to consider.

The school secretary has the job of checking to see if all semester grades have been entered. Ms. Epyt checks to see if the number of grades matches the number of courses. Help her out by writing a program to keep track of how many school grades a particular youngster has. We need a DATA statement which includes a complete list of student names and grades.

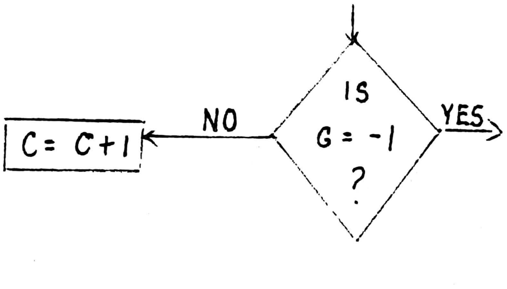
In solving this problem, we have the difficulty of anticipating the number of grades a youngster may have. One might have 5, another 6, another 2. Hence, we need to be flexible in counting. Additionally, we are forced to read one grade at a time. As we read a grade we will count it.

Before flowcharting the solution, a word about this counting technique. What we construct will be a COUNTER, not a COUNTING LOOP. The COUNTER will have an initialization and an increment, but it will not have a test and transfer. There is really nothing to test against, because the total number of grades for each student is unknown. So this is not a true loop, and is referred to simply as a COUNTER.

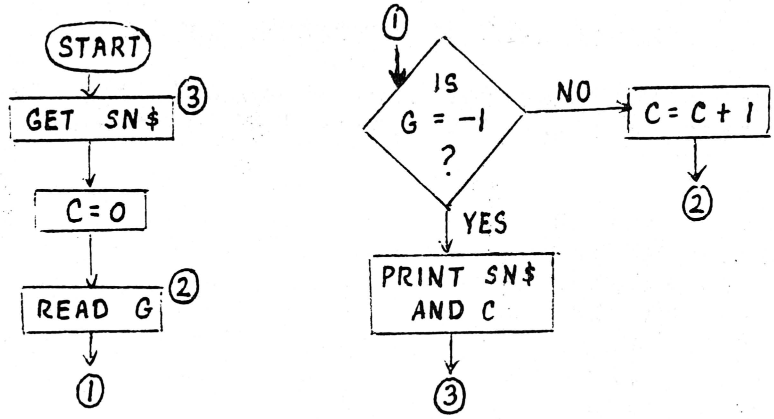
Let's begin our flowchart. In our data, a name is followed by a series of grades. Therefore we must read the name separately from the grades. Also, we certainly do not want to start our counter until the name is read.



In our solution, we now want to increment the counter and read the next grade. But how will we stop the counter once we complete a set of grades? Once again we will have to use a piece of "dummy data". We will insert a piece of dummy data between the end of one set of grades and the next student's name. We can then test for that piece of dummy data, before incrementing. Here we have used -1 as the dummy.



Once the dummy data, -1, is read, the complete set of grades for one student has been counted. And sure enough, we have the number of grades stored under C, or our counter. We can now output the student's name and number of grades. The process can then begin again with the next student's name. And as the second instruction indicates, we will reinitialize the counter at 0.



The program and its execution is shown below. Look carefully at the DATA statement. A terminating flag was also added.

```
10 READ SN$
15 DATA "JONES", 65, 76, 80, -1, "SMITH", 89, 93, -1
16 DATA "BROWN", 99, 92, 53, 100, -1, "STOP"
18 IF SN$ = "STOP" THEN 90
20 LET C = 0
30 READ G
40 IF G = -1 THEN 70
50 LET C = C + 1
60 GO TO 30
70 PRINT S$;" HAS"; C;" SCHOOL GRADES."
80 GO TO 10
90 END
```

```
JONES HAS 3 SCHOOL GRADES.
SMITH HAS 2 SCHOOL GRADES.
BROWN HAS 4 SCHOOL GRADES.
```

The looping process is effective for repetitive patterns within a program. Here we have seen the loop allowed to count the repetitions. Depending on the task, we can control the number of loops -- COUNTING LOOP -- or we can keep a running count of the number -- COUNTER.

EXERCISES 3.3

1. A dash of that, a pinch of this, and soon you need the Galloping Gourmet to keep track of the ingredients. Write a program which counts the number of different ingredients in a given recipe. Allow the user to input each ingredient one at a time. The output might look like this:

```
PAELLA USES 49 DIFFERENT INGREDIENTS
```

2. A constellation just for you -- wouldn't that be something? Write a program to print out the "T" constellation -- which just appeared on the horizon.

```
XXXXXXX
XXXXXXX
XXXXXXX
XXXXXXX
  XXX
  XXX
  XXX
  XXX
  XXX
  XXX
  XXX
```

Use one counting loop for the top of the T, and one for the bottom portion.

3. Set up a program for asking the $64,000 question. Ask the following riddle to the keyboard player: If seven maids with seven mops sweep seven tons of sand in seven months, how long does it take ten maids with ten mops to sweep ten tons of sand?
He/she has four chances to answer correctly. If they do answer correctly, award the money. If not, have the audience make appropriate "hissing".

4. Trace the following "READER" program. Be sure to include a table of values and the output.

```
10 FOR X = 1 TO 3
20 READ A
25 IF A = 999 THEN 999
30 NEXT X
35 PRINT A
40 GO TO 10
50 DATA 2,7,8,16,14,9,25,-1,6,5,3,17
55 DATA 20,15,17,10,1,22,0,999
999 END
```

5. Write a program to simulate a voting booth tally machine. The voters enter the booth with a choice of two candidates:

#1 SPIRIT AGNES
#2 RICHARD NIXMAN

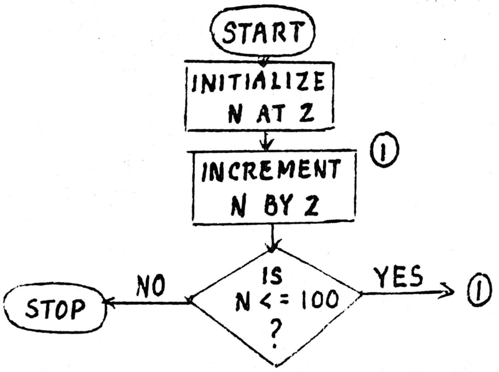
You'll need two counters, one for each candidate. As each vote -- 1 or 2 -- is inputted, your program will need to branch to the appropriate counter. After all the votes are in, print out the number of votes for each candidate.

SECTION 3.4 ACCUMULATING LOOPS

Using loops, we were able to develop counting techniques. Using the counting loop and the counter we can keep track of loop repetitions. We can also use looping techniques to accumulate numbers. Our next problem will demonstrate this capability.

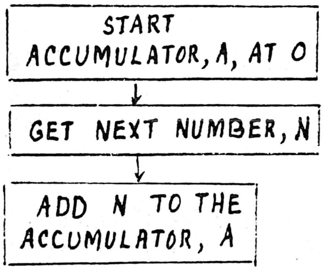
Write a program to find the sum of the even numbers from 2 to 100.

Our solution to this problem will not resemble the way we would find such a sum. We would simply stack the numbers and add them up in continuous columns. But in the Apple solution we have to consider the way numbers are generated. In a loop to produce the even numbers, each new number replaces the former number in the mailbox. There is no way to hold on to all the evens generated. Take a look at a flowchart for creating the evens. As you step through it, verify that each number is lost as the loop continues.

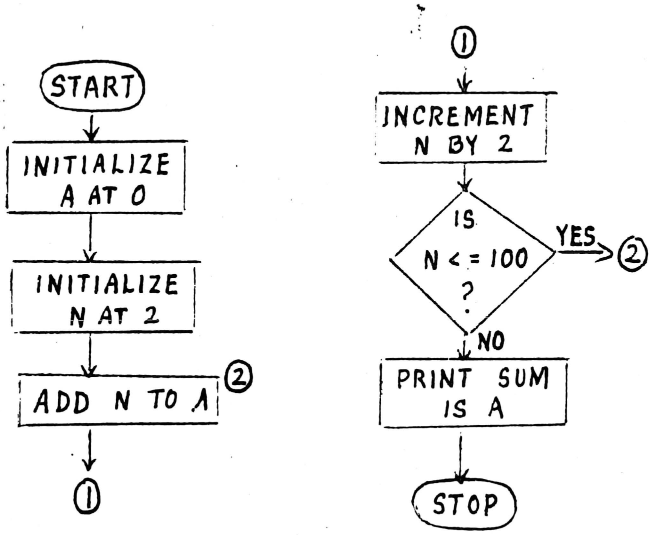


Therefore, into this loop we must keep a running total of these numbers. Each time a new number is generated, that number must be added into that running accumulator. By the time N = 100, we should have added 2 + 4 + ...+ 98. How do we create this accumulator? At what value should it be initialized? What increment value would it have? Do we need to test that running sum? Let's dive in and tackle these questions.

An accumulator would certainly need to start without any value in it. Hence it must be initialized at 0. Each time a new number is created, that number needs to be applied to the total. The increment for this total must be each new N number. There is no way for us to predict what that total will be. Then, of course, there will be no test for the accumulator. Like the counter, this will be a partial loop. It's flowchart would look like:



To solve the problems we need to combine our even number generator with the accumulator. We need to be certain to keep the accumulator initialization outside the other loop. We do not want to reset the total at 0 each time we create a new even number. We must also be careful not to add a number to our total until we are certain it has passed the test: IS N < = 100? Our flowchart solution follows.



Before writing the program, a word about the increment statement for the accumulator, A -- ADD N TO A. Like each increment it is a LET statement. Within that statement we want to reassign the value stored in A. The new value will be the new number added to the value presently in A.

```
LET A = A + N
```

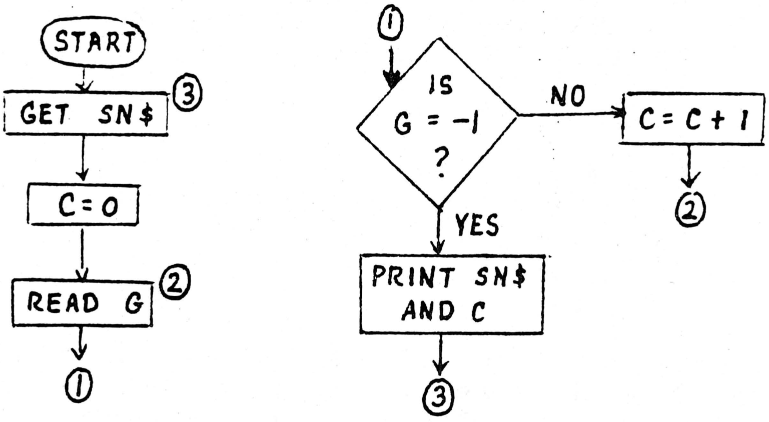
Below we have written the program from the flowchart. You should certainly try your own before glancing at ours. Think! You will see that we have used a traditional loop in the left program and a FOR-NEXT loop in the one on the right. Take time to trace each, if you are having any difficulty.

```
10 LET A = 0                           10 LET A = 0
20 LET N = 2                           20 FOR N = 2 TO 100 STEP 2
30 LET A = A + N                       30 LET A = A + N
40 LET N = N + 2                       40 NEXT N
50 IF N < = 100 THEN 30                50 PRINT "THE SUM OF THE"
60 PRINT "THE SUM OF THE"              60 PRINT "FIRST ";N
70 PRINT "FIRST ";N                    70 PRINT "NUMBERS IS: ";A
80 PRINT "NUMBERS IS: ";A              80 END
90 END
```

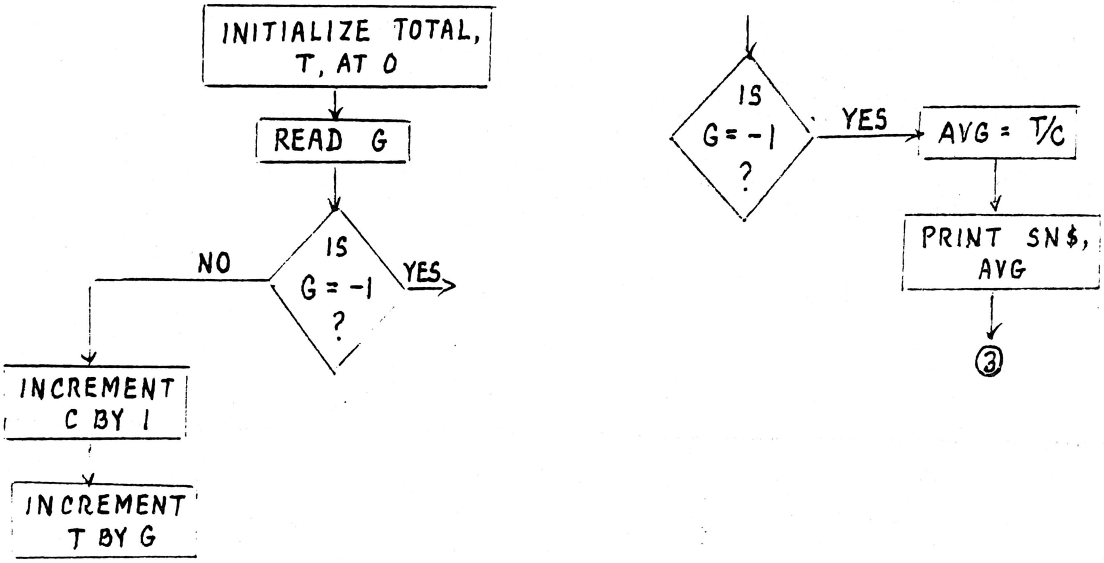
As a summary of the counting and accumulating techniques, we would like to return to a former program and add something to it.

In the last section we wrote a program to help the school secretary by counting the number of grades for a particular student in the class. The principal suggested that while we are at it, she would like to have access to each student's average. So to our program we would like to add an accumulator that would allow us to total the grades for each student. With the count and the total, what will we be in a position to do? If you said "average", you are right with us! So our new project is to find the average of each student's grade.

Here is the flowchart from the counting problem:


START
GET SN$
③
C=0
READ G
②
①
①
IS G = -1 ?
NO
C=C+1
②
YES
PRINT SN$ AND C
③

To this we will need to work in the accumulator. We need to initialize this total, prior to reading in the grade. And assuming the flag value is not incurred, we will add the grade, G, to this total. We have added those parts to the flowchart section on the left.


INITIALIZE TOTAL, T, AT 0
READ G
IS G = -1 ?
NO
YES
INCREMENT C BY 1
INCREMENT T BY G
IS G = -1 ?
YES
AVG = T/C
PRINT SN$, AVG
③

Once the flag is encountered and G = -1, then we are ready to find the average using C and T. The YES path is demonstrated on the right above.

Our new flowchart has a counter to record the number of grades and an accumulator to find the sum of these grades. Two partial loops are interwoven with the grade reading loop. The final program looks like this:

```
10 READ SN$
20 LET C = 0
30 LET T = 0
40 READ G
45 DATA "JONES", 65, 76, 80, -1, "SMITH", 89, 93, -1
47 DATA "BROWN", 99, 92, 53, 100, -1, "STOP"
50 IF G = -1 THEN 80
60 LET C = C + 1
   LET T = T + G
70 GO TO 40
80 LET AVG = T/C
90 PRINT SN$;" HAS AN AVERAGE OF "; AVG
100 PRINT
110 GO TO 10
120 END
```

We have also included a trace of the program. Note the re-initialization of both C and T each time a list of grades is complete. Also make a special note of how the accumulator T adds the grades.

	STORAGE	VALUES		
SN$	C	T	G	AVG
JONES	0	0	65	
	1	65	76	
	2	151	80	
	3	231	-1	77
SMITH	0	0	89	
	2	182	-1	91
BROWN	0	0	99	
	1	99	92	
	2	191	53	
	3	244	100	
	4	344	-1	86

OUTPUT

```
JONES HAS AN AVERAGE OF 77

SMITH HAS AN AVERAGE OF 91

BROWN HAS AN AVERAGE OF 86
```

By the way, the principal told us to tell you to stop fooling around and get to the exercises!

EXERCISES 3.4

1. Translate each of the following statements into Applesoft.

A.) Add N to the existing sum S.

B.) Add 1 to the counter C.
C.) Initialize the sum at 0.
D.) Loop the counter from 0 to 50.

2. The program below finds the sum of the first one hundred counting numbers. It is often a better program if it has a more general application. Make any and all modifications in order to allow the user to input the boundaries for the set to be accumulated. The output should say:

```
THE SUM OF THE NUMBERS FROM 20 TO 50 IS 1050.
```

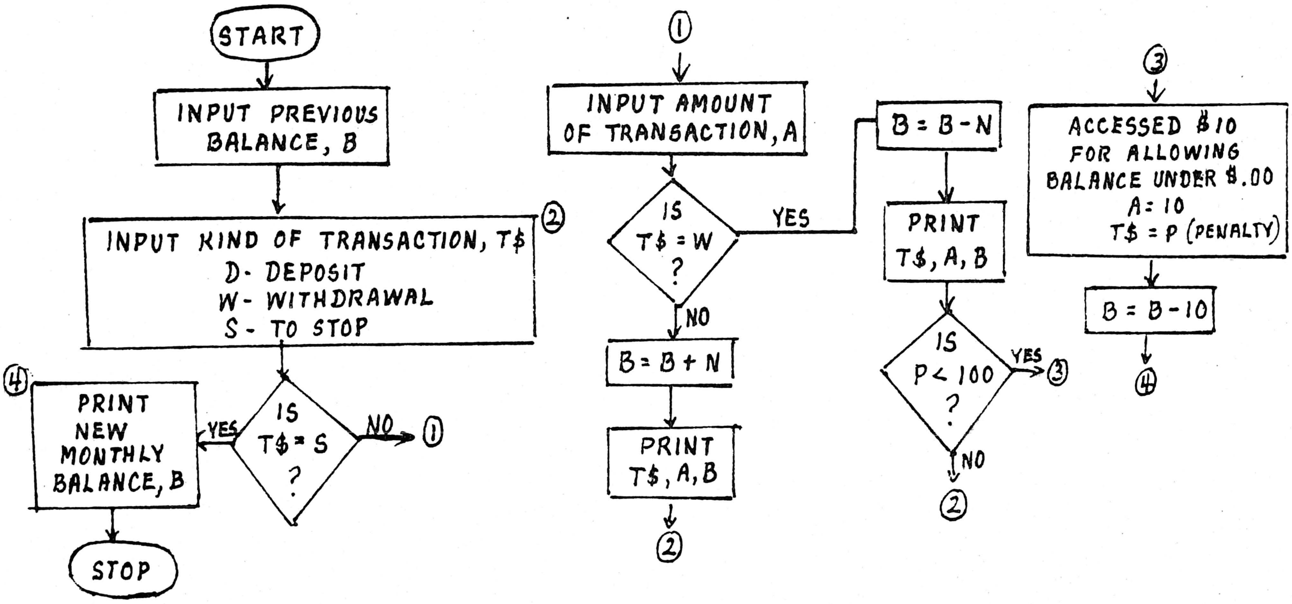
```
5 LET S = 0
10 FOR X = 0 TO 100
20 LET S = S + X
30 NEXT X
40 PRINT "SUM IS";S
```

3. This symbol -- ! -- means excitement. But in the study of probability and chance, this symbol is used for special products called FACTORIALS. Five factorial 5! represents the product of the first five counting numbers.

EXAMPLES: 5! = 1*2*3*4*5 = 120
3! = 1*2*3 = 6
7! = 1*2*3*4*5*6*7 = 5040

Use a product accumulator to write a factorial producer program. Allow the keyboard user to input a value N and to have the Apple output N! Be careful, factorials get very large quickly. You might need to give instructions for a maximum range.

4. With your ever-growing financial status, you will soon have your own checking account. You can have the Apple assist you in maintaining your records. Below is a flowchart for keeping a monthly record of deposits and withdrawals to find your current checking balance. Translate the flowchart into a program. Be sure to have a complete output. Also be sure to include all paths.


START
INPUT PREVIOUS BALANCE, B
② INPUT KIND OF TRANSACTION, T$
D- DEPOSIT
W- WITHDRAWAL
S - TO STOP
IS T$ = S ?
YES
NO
①
④ PRINT NEW MONTHLY BALANCE, B
STOP
①
INPUT AMOUNT OF TRANSACTION, A
IS T$ = W ?
YES
NO
B = B + N
PRINT T$, A, B
②
B = B - N
PRINT T$, A, B
IS P < 100 ?
YES
③
NO
②
③
ACCESSED $10 FOR ALLOWING BALANCE UNDER $.00
A: 10
T$ = P (PENALTY)
B = B - 10
④

5. At the carnival the favorite ride this year is the "BENDER". It involves a 360 degree rotation of a cage full of bodies. Each cage can safely hold up to 1000 pounds. As each person enters the ride he/she is weighed and a running total is kept. As soon as the weight total goes over 1000 pounds, a light flashes and the person who tipped the scale is told to wait for the next ride. Write a program to simulate the "BENDER" weight accumulator. Read in a list of weights. Accumulate the total, giving appropriate instructions to each participant. When the ride is full, begin the process again. Don't forget the unlucky holdover from the last ride!

SECTION 3.5 COMPLETELY LOOPED

We have previously explored the FOR-NEXT statements and the way in which they are combined to produce a looped repetition. Here we will see how two or more FOR-NEXT loops can be combined. Applesoft provides for constructing loops within loops. Such an arrangement is called NESTED LOOPS. Nested loops will allow us to solve problems of greater complexity.

Our programming problem is to calculate compound interest. The First United Bank is running a special advertisment for a new investment plan. Invest any amount over $1000 for a three year period and you can earn 15% per year interest compounded quarterly. Sound good? Write a program that will allow an inputted investment and output the amount of money returned at the end of the three years.

The notion behind using a nested loop is that we have a given repetitive task, itself repeated several times. In our problem, the interest is compounded several times during the year. For example, quarterly compounded means that four times a year the interest is calculated and applied to the invested money. If the investment is for three years then the four-fold compounding is repeated three times. In all, the interest is compounded twelve times. The idea of one loop within another provides that each trip through the outermost loop, the inside loops goes through its complete cycle of loops.

Let's look at some examples before we get back to our problem.

```
10 FOR A = 1 TO 4
20 FOR B = 1 TO 3
30 PRINT "A="A ; "B" = B,
40 NEXT B
50 NEXT A
60 END
```

In this program, the loop of the variable B is contained within the A-loop. For each loop of A, the B loop will complete all of its loops. Check the output below:

```
A=1 B=1  A=1 B=2  A=1 B=3
A=2 B=1  A=2 B=2  A=2 B=3
A=3 B=1  A=3 B=2  A=3 B=3
A=4 B=1  A=4 B=2  A=4 B=3
```

The first line of output shows an A value of 1. Then the program enters and completes the B loop, three times. After B=3, the program exits the B loop but reenters the A loop. A is incremented to 2 and the B loop is initialized back to 1. Again the B loop goes to its limit. Line 30 is executed a total of 12 times. Note the use of the column-maker comma. After filling the three columns, each a new B value, the printer proceeds to a new line and continues with the columns.

Another program will help your understanding of nested loops. This program is used for printing out a portion of the multiplication table. The execution is shown on the right.

```
10 FOR X = 2 TO 4
20 FOR Y = 10 TO 12
30 PRINT X "*" Y "=" X * Y
40 NEXT Y
50 NEXT X
60 END
```

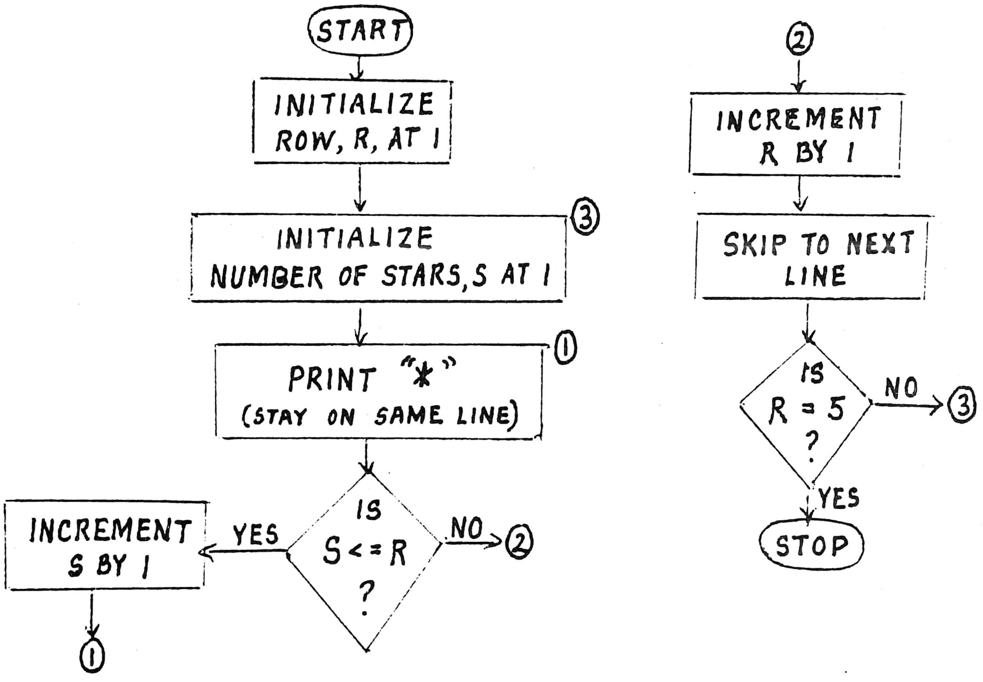
```
2 * 10 = 20
2 * 11 = 22
2 * 12 = 24
3 * 10 = 30
3 * 11 = 33
3 * 12 = 36
4 * 10 = 40
4 * 11 = 44
4 * 12 = 48
```

Line 10 initializes X's value at 2. X will retain that value until line 50. Before line 50, the computer will run through the Y loop, line 20-40, a total of 3 times. When the Y loop has finished three passes, X is incremented and the program returns to line 20. At this point, Y is again initialized at 10 and the process continues. Y will take on its values 10, 11, and 12 three times.

It is possible within nested loops to control the inner loop limits with the outer loop. The inner loop will have a variable test. Below is a quick program to do the following:

```
*
**
***
****
*****
```

We want five rows of stars. Each row has an increasing number of *'s printed out. Therefore each inner loop occurs as many times as the row indicator.



START
INITIALIZE ROW, R, AT 1
INITIALIZE NUMBER OF STARS, S AT 1
③
PRINT "*" (STAY ON SAME LINE)
①
IS S <= R ?
YES
NO
②
INCREMENT S BY 1
①
②
INCREMENT R BY 1
SKIP TO NEXT LINE
IS R = 5 ?
NO
③
YES
STOP

Try to translate this into Applesoft before you look at our program below. The number of stars loop is within the row counter loop. Be careful to use the trailing punctuation. You'll need a dummy PRINT statement to void the trailing semicolon at the end of each row.

```
10 FOR R = 1 TO 5
20 FOR N = 1 TO R               *
30 PRINT "*" ;                  **
40 NEXT N                       ***
50 PRINT                        ****
60 NEXT R                       *****
70 PRINT "TAH DAH"              TAH DAH
80 END
```

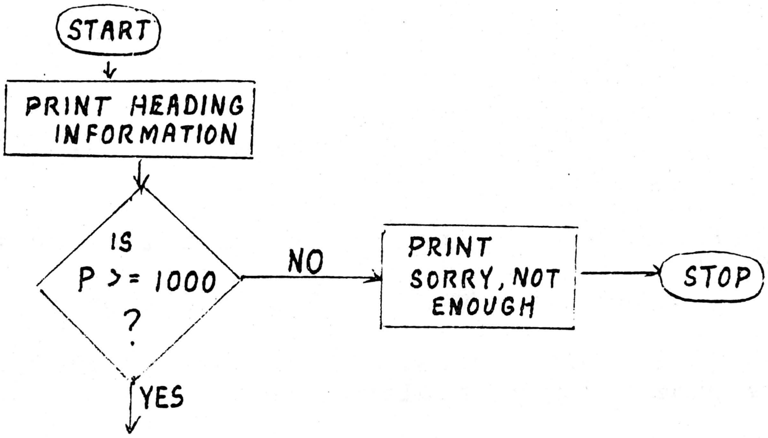
The rule to remember when using nested loops is:

NEVER OVERLAP LOOPS

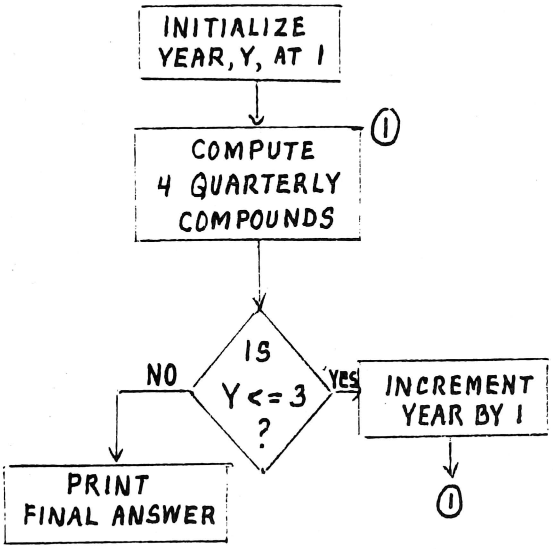
Each inner loop must be totally contained within the other loops. The program on the left demonstrates the correct nesting of loops. The two nested loops on the right cross. The second program returns an error: NEXT WITHOUT FOR ERROR

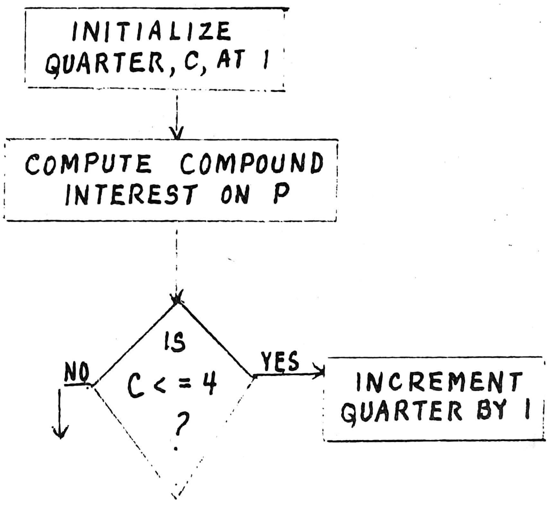
```
10 FOR A = 1 TO 2               10 FOR A = 1 TO 3
15 PRINT "NESTED" ;             15 PRINT "NESTED" ;
20 FOR B = 1 TO 3               20 FOR B = 1 TO 2
30 PRINT "LOOPS"                30 PRINT "LOOPS"
40 NEXT B                       40 NEXT A
50 NEXT A                       50 NEXT B
```

It is time to return to our compound interest problem -- compounding quarterly for a three year period. Let's construct a flowchart together. We begin with some explanatory information and proceed to allow the user to input the amount of his investment -- called principal. We might check the amount, for it must be at least $1000.



With that out of the way let's consider our two nested loops. One loop will be the three year repetition. The inner loop will be the compounding loop -- executed four times. In case you have trouble with compounding, that means calculating the quarterly interest and applying it to the current principal (quarterly interest equals one-fourth annual interest). That new amount -- principal + interest -- becomes the new principal. Here are the two loops separated:


INITIALIZE YEAR, Y, AT 1
COMPUTE 4 QUARTERLY COMPOUNDS
①
IS Y <= 3 ?
NO
YES
INCREMENT YEAR BY 1
①
PRINT FINAL ANSWER


INITIALIZE QUARTER, C, AT 1
COMPUTE COMPOUND INTEREST ON P
IS C < = 4 ?
NO
YES
INCREMENT QUARTER BY 1

Next, we need to insert the compounding loop within the year loop. Our program looks like this:

```
10 PRINT "THIS PROGRAM WILL FIND"
12 PRINT "QUARTERLY COMPOUNDED"
14 PRINT "INTEREST ON A THREE"
16 PRINT "YEAR INVESTMENT"
18 PRINT
20 INPUT "INPUT INVESTMENT (AT LEAST $1000): ";P
25 IF P > = 1000 THEN 30
27 PRINT "SORRY, NOT ENOUGH"
28 GO TO 90
30 FOR Y = 1 TO 3
40 FOR C = 1 TO 4
50 LET P = P + P * .15/4
60 NEXT C
70 NEXT Y
80 PRINT "YOUR INVESTMENT YIELDS: "$" ; P
90 END
```

With an input of $1000, the final output from line 80 is:

```
YOUR INVESTMENT YIELDS: $1555.45433
```

You made $555.45 in three years. E. F. Hutton says...... "That appears to be a sound investment." Congratulations!

EXERCISES 3.5

1. Trace the following NESTED LOOP program with a table of STORED VALUES and the properly formatted output.

```
10 FOR L = 1 TO 3
20 PRINT OUTER LOOP # ";L
30 FOR N = 1 TO 5
40 PRINT "*" ;
50 NEXT N
60 NEXT L
70 END
```

2. As the Apple representative for your school, you want to produce advertising labels. Each label looks like this:

```
XXXXXXXXXXXX
APPLE
 APPLE
  APPLE
   APPLE
    APPLE
     APPLE
      APPLE
       APPLE
XXXXXXXXXXXX
```

Write a program to produce fifty such labels.

3. Below are three looping programs. One of these has an error which will prevent its execution. Identify the problem which is incorrectly designed and explain the necessary correction.

A.)

```
10 FOR X = 1 TO 10
20 PRINT X
30 FOR Y = 1 TO 3
40 PRINT Y
50 NEXT Y
60 NEXT X
```

B.)

```
10 FOR X = 5 TO 1 STEP -1
20 FOR Z = 1 TO 3
30 PRINT X-Z
40 NEXT X
50 NEXT Z
```

C.)

```
10 FOR A = 1 TO 5
20 PRINT A
30 NEXT A
40 FOR B = 1 TO 3
50 PRINT A * B
60 NEXT B
```

4. The 80's brought a "RETURN TO THE BASICS" move back to school. Everyone remembers memorizing the addition table. Use nested loops to print out a complete addition table for the numbers 0 to 9. It might look like this:

```
+ 0 1 2 3
0 0 1 2 3
1 1 2 3 4
2 2 3 4 5
3 3 4 5 6
```

Print out the column headers before entering the looping routines. The row operators can be printed out inside the loop. Think through the tabbing mechanism.

A LITTLE BIT MORE SECTION 3

SPICING UP THE PRINT STATEMENT

By this time you should have a good understanding of how to use the PRINT statement and all the intricacies involved with punctuation. Normally, the PRINT statement output is a white letter on a black background. It is possible however to alter this appearance. We can change this output to the inverse, a black letter on a white background. The statement to do this is INVERSE. Look at the following program:

```
10 INVERSE
20 PRINT "HELLO THERE!"
```

The result is the string HELLO THERE! But now the letters are black and the background directly behind is white. Note that only the outputted characters are in the inverse condition, the rest of the screen is normal. Once the INVERSE statement is executed, all output will be in this black on white format.

In order to revert back to the normal white on black condition, we must use another command, the NORMAL statement.

```
40 NORMAL
```

This statement, once executed, will result in the typical output format.

If this inverse situation isn't flashy enough for you, we can provide for more. Through the use of the FLASH command, it is possible to have the output flash from normal to inverse to normal to inverse, etc.

```
10 FLASH
20 PRINT "HELLO THERE"
```

In the above program, the output HELLO THERE goes from black on white then to white on black, similar to the flashing cursor. Once again, all output will be flashing until a NORMAL statement is executed. Note that the FLASH and INVERSE will only affect output and will not change the appearance of characters in a listing or entered at the keyboard.

The following program is one to demonstrate our fancy footwork with output. Our students call it "STARRY NIGHT". It consists of a loop which outputs "*" in various locations read into the program, and in various forms. In the READ statement, X and Y are the location, Z controls the format.

```
10 HOME
20 FOR A = 1 TO 10
30 READ X,Y,Z
40 DATA 2,18,1,6,38,2,22,12,3,19,19,1
41 DATA 24,24,2,10,1,2,1,40,3
43 DATA 9,15,1,6,6,2,22,28,1
45 VTAB X
46 HTAB Y
50 IF Z = 1 THEN 100
60 IF Z = 2 THEN 130
70 FLASH
80 PRINT "*"
90 GO TO 150
```

```
100 INVERSE
110 PRINT "*"
120 GO TO 150
130 NORMAL
140 PRINT "*"
150 NEXT A
```

Design your own constellations. Enjoy all the possibilities and consider working them into your solution.

REVIEW EXERCISES

CHAPTER THREE

1. Trace the following program with a table of STORED VALUES and properly formatted OUTPUT.

```
10 PRINT "IF JAN 1 IS A SUNDAY THEN..."
20 FOR W = 0 TO 4
25 FOR D = 7 * W + 1 TO 7 * (W + 1)
30 PRINT D; SPC(3);
40 IF D = 31 THEN 60
50 NEXT D
60 PRINT
70 NEXT W
80 END
```

2. Write a program which will scroll up the lines until the screen is completely cleared. (See A Little Bit More Chapter 2 for help)

3. Write a program which will read in a person's first and last name as a single string and print it out last name, first name.

CHARLES BABBAGE becomes...BABBAGE, CHARLES

4. Translate each of the following statements to Applesoft.

A.) Initialize X at 100
B.) Loop V for 2 to 20
C.) Increment Z by three's
D.) Add each new N to the sum, S
E.) When X is larger than 20, end the loop (otherwise continue at line 10)
F.) Transfer to line 10 when the string is Yes or Yep.
G.) Add 1 to the counter C
H.) Test and transfer back into the X loop
I.) Loop D from 0 to 100 decreasing it by 1 each time
J.) Transfer to line 60

5. The carnival ride program from Section 3.4 is really incomplete. Modify this program to allow for the following changes.

A.) There are six cars filled for each ride. Keep track of how many cars are filled. When the sixth car is full, hold up the line while the ride is going.

B.) Allow for the unloading and restarting of the ride. Repeat the whole process a certain number of times.

C.) Give complete information and directions to each passenger.

CHAPTER 4

OPENING NEW DOORS

Section 4.1 The Great Chopper
Section 4.2 GOZINTA!
Section 4.3 Stroll Through Number Theory
At The Keyboard: Peeking Into The Future
Section 4.4 A Program Per Chance
Section 4.5 Gaming Randomly
A Little Bit More: Economy Size
Review Exercises

SECTION 4.1 THE GREAT CHOPPER

Much can be said about the functions that the Apple is capable of performing. We have discussed mathematical functions like ABS, string functions like MID$, and spacing functions like TAB. But our investigation now turns toward a very special function and how we can use it creatively.

Blaise has upgraded his robot -- FUNCTION ONE -- to perform a new and quite special function. Below is a table of the inputs and outputs for this new function. Look carefully at it and see if you can determine what is occurring.

INPUT	OUTPUT
2	2
3.475	3
11.69	11
.35	0
-1.75	-2
-3	-3
-4.1	-5

Have you guessed what this new function is doing? If you said rounding off the input to the nearest integer, you are getting warm. However, 11.69 would have been rounded to 12. If you said we were chopping off any decimal part of the input, you are even hotter. But, if that was the case, -1.75 would have been chopped to -1. With these clues, perhaps you've come to the conclusion that we are "rounding down" to the next smallest integer.

Let us help you put it into words. For every input, the corresponding output is the greatest integer which is less than or equal to the input. We call this function GREATEST INTEGER FUNCTION. The notation used is INT; and it has a single argument.

Let's check our values to see if this is really happening. Is 3 the greatest integer less than or equal to 3.475? Is 0 the greatest integer that is still less than or equal to .35? Is -1 less than or equal to itself? What integer is less than or equal to -4.1? Is it -5?

Pretend you are the Apple and compute the GREATEST INTEGER FUNCTION for each of the following problems:

A.) INT(12.78)
B.) INT(3.12)
C.) INT(14)
D.) INT(.98)
E.) INT(-2)
F.) INT(-3.8)
G.) INT(-9.14)
H.) INT(-.375)
I.) INT(0)
J.) INT(SQR(26))

ANSWERS: 12,3,14,0,-2,-4,-10,-1,0,5

No doubt you might be asking yourself why would the Apple have a function like this, instead of a rounding off function? How did anyone come up with this idea? Well, the INT function is part of Applesoft because it is more powerful than a rounding off function. Not only can we use the INT for rounding, but it can be used for other purposes as well.

Here we are going to explore how INT can be used to round off. In doing this, we want to look at this programming problem.

Often when we work with dollars and cents, our calculations generate money values such as $1555.4533. This was the output of our compounding problem in Chapter 3. Obviously, when we speak of money, only two decimal places are significant. A penny is our smallest denomination and hence we would like the value rounded off to the nearest hundredth place -- $1555.45. Write a program to allow the user to input an amount of money and round it off to the nearest cent.

Before beginning, round off each of the following numbers to the nearest hundredth, so that we can analyze the technique.

A.) 13.478
B.) 4.373
C.) 9.6958
D.) 1555.4533

To round off, we must look at the number to the right of the hundredth place. That number in the thousandths place becomes an indicator. If it is 5 or greater, we add 1 to the digit in the hundredths place. If it is 4 or less, the hundredths place digit remains the same. Then, in both cases, we chop off the remaining digits. Here are the correct answers:

A.) 13.48
B.) 4.37
C.) 9.70
D.) 1555.45

Below we are going to introduce and explain the procedure to use with the Apple. With each step we have given two examples. Read these carefully:

A.) Multiply the number by 100. This moves the decimal place two places to the right.

13.478 * 100 = 1347.8 4.373 * 100 = 437.3

B.) Add .5 to the number. This step replaces our ability to see if the thousandths place digit is above or below 5. Note the different effect in the hundredths place digit in these numbers.

1347.8 + .5 = 1348.3 437.3 + .5 = 437.8

C.) Take the INT of this value. That eliminates all the remaining digits after the hundredths place digit.

INT(1348.3) = 1348 INT(437.8) = 437

D.) Divide by 100 to move the decimal point back to its original position, two places to the left.

1348/100 = 13.48 437/100 = 4.37

We have rounded off each number appropriately. Now let's have the Apple do this, one step at a time. Note that we have instructed it to print out the value of M after each step, so that we can see the process occurring.

```
10 LET M = 1555.4533
20 PRINT M
30 LET M = M * 100
40 PRINT M                         1555.4533
50 LET M = M + .5                  155545.33
60 PRINT M                         155545.83
70 LET M = INT(M)                  155545
80 PRINT M                         1555.45
90 LET M = M/100
100 PRINT M
```

The same result can be accomplished in a single statement.

```
10 LET M = 155.4533
20 PRINT INT(M * 100 + .5)/100
```

We've taken special care to do the operations in the correct order. To complete our problem, our program might look like this:

```
10 INPUT "INPUT MONEY AMOUNT:" ;M
20 LET R = INT(M * 100 + .5)/100
30 PRINT " ";M; "ROUNDED OFF IS $";R
40 GO TO 10
```

The same technique can be used for rounding off to any power of ten. The only part that changes is what factor we multiply and divide by. If we wished to round off to the nearest hundred thousandth, then we will multiply and divide by 10 or 100,000.

```
20 LET R = INT(M * 10 ^ 5 + .5)/10 ^ 5
(OR INT(M * 100000 + .5/100000)
```

To round off to the nearest hundred, the process is altered by the direction we wish to move the decimal point. To begin with, we would wish to move the decimal two places to the left (as opposed to moving it to the right). Therefore we must multiply by 10 or divide by 100. And at the end of the procedure, we would move the decimal back to the right two places. That means dividing by 10 or multiplying by 100. Here is the statement for rounding to the nearest hundred.

```
10 LET R = INT(M * 10 ^ -2 + .5)/10 ^ -2
(OR INT(M/100 + .5) * 100)
```

The ability to round off is a procedure that can function on its own or can be built into other programs for the purpose of rounding off inputs and outputs. And even though INT function does not itself round off, it gives us the potential for rounding.

EXERCISES 4.1

1. Find the value of each expression below.
A.) `INT(-6.9)`
B.) `INT(1.4 ^ 2)`
C.) `INT(2 + 3/6)`
D.) `INT(SQR(50))`
E.) `INT(ABS(-4.2))`
F.) `.5 * INT(-2.1)`
G.) `INT(100 * 6.12)/100`

2. Write an Applesoft statement which stores in R the value of N when:
A.) N is to be rounded to nearest thousandth.
B.) N is to be rounded to nearest whole number.
C.) N is to be rounded to nearest ten.

3. Trace the following program with a table of stored values and the output.

```
10 READ N
20 DATA 638,702,915,0
30 IF N = 0 THEN 100
40 LET D1 = INT(N/100)
50 LET X = N - D1 * 100
60 LET D2 = INT(N/10)
70 LET D3 = X - D2 * 100
80 PRINT N " = ";D1;"*10 ^ 2 + ";D3; * 10 ^ 0
90 GO TO 10
100 END
```

4. It is a wondrous feeling to sit back and watch money grow. On Januar 1, 1980, Blaise invested $100 in a savings account, paying interest at 11 1/2% per year, compounded annually. Write a looping program to find the amount accumulated after each of the twelve years. Here is a sample output.

```
JANUARY 1 1980 $100
JANUARY 1 1981 $111.50
JANUARY 1 1982 $124.32
```

Be sure to round off dollar amounts appropriately.

5. Using the LEN function, it is easy to find out the number of characters in a string. Finding the number of digits in a number is a much harder task. Write a program that will take any whole number and count the number of digits in the number. The output should say:

```
6438 HAS 4 DIGITS
17506 HAS 5 DIGITS
```

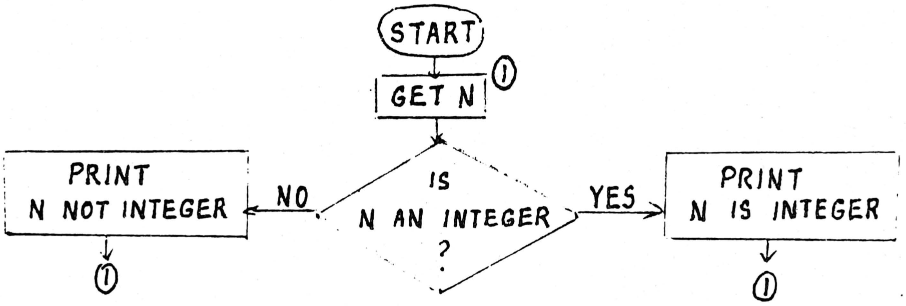
SECTION 4.2 GOZINTA!

Our next Section will help prove our point about the INT function. Although rounding off is a nice tool to have, the power of the INT function goes well beyond this simple capability. The programming problem will demonstrate.

The fifth grade math classes have been studying the DIVISIBILITY RULES. The students know that a number is divisible by 2 if it's even; divisible by 5 if it ends in 0 or 5; divisible by 3 if the sum of the digits is a multiple of 3. They even have rules for divisiblity by 9 or 11. But they still have a problem with 7. When does 7 divide evenly into a number? As a student tutor, you decide to help the situation. Write a program that will tell if a number is divisible by 7.

Prior to beginning our solution to this problem, we would like to ask a question related to divisibility. The question: How can you recognize an integer from other rational numbers? For you, it probably seems simple. You just need to see if there is a fractional or decimal part to the number. Without such a part, you would come to the conclusion that the number was an integer. But how about the Apple? It cannot just look for a decimal point. And the process becomes a harder discrimination test.

Together we would like to write a short program for distinguishing integers from non-integers. A branching program is a necessity. The flowchart might look like this:



But we still have our problem of how to translate the decision box -- IS N AN INTEGER? To do this, we need to employ our INT function once again. Complete these two sets of problems and do so while remembering our question.

A.)	INT(24)	A.)	INT(3.4)
B.)	INT(-4)	B.)	INT(-1.2)
C.)	INT(2)	C.)	INT(16.28)
D.)	INT(0)	D.)	INT(.9)

Let's see if we draw some conclusions. Certainly the four arguments on the left are integers. The four on the right are not. And for the four left ones the INT of the argument number is the same as the argument. Is that true for the others? Therefore we can conclude: INT(X) = X, only when X is an integer. This is our test:

IF INT(X) = X THEN ____

Our program should be easy to write now. Notice that we have used READ-DATA statements. The output is on the right.

```
10 READ X
20 DATA 3.4,5,2.78
25 DATA -9,-2.7,99                     3.4 IS NOT AN INTEGER
30 IF X = 99 THEN 80                   5 IS AN INTEGER
35 IF INT(X) = X THEN 60               2.78 IS NOT AN INTEGER
40 PRINT X; "IS NOT AN INTEGER"        -9 IS AN INTEGER
50 GO TO 10                            -2.7 IS NOT AN INTEGER
60 PRINT X; "IS AN INTEGER"
70 GO TO 10
80 END
```

Now what does this program have to do with our divisibility problem? How do we decide if a number is divisible by 7? That means that 7 divides into the number such that the quotient is an <u>integer</u>. 7 GOZINTA the number evenly without a remainder. Therefore we can use our test for discriminating an integer from a non-integer. But we need to look not at number X, but the quotient X/7.

Look at this set to understand our direction.

A.)	28/7	A.)	INT(28/7)
B.)	10/7	B.)	INT(10/7)
C.)	84/7	C.)	INT 84/7
D.)	102/7	D.)	INT(102/7)

For examples A.) and B.), the remainder for 28/7 must be zero, for the INT function does not change the value at all.

So we can ask the Apple, "Is a number N is divisible by 7?", by instructing it to look at this statement:

```
30 IF INT(X/7) = X/7 THEN _____
```

7. Our program evolves into the following:

```
10 READ X
20 DATA 3.4,5,2.78
25 DATA -9,-2.7,99
30 IF X = 99 THEN 80
35 IF INT(X/7) = X/7 THEN 60
40 PRINT X; " IS NOT DIVISIBLE BY 7"
50 GO TO 10
60 PRINT "7 GOZINTA ";X
70 GO TO 10
80 END
```

So easy is this notion of "going in evenly" that we have a statement for any divisibility checks.

A.) Does 17 go into 87342 evenly?
IF INT(87342/17) = 87342/17 THEN _____

B.) Is M divisible by 4?
IF INT(M/4) = M/4 THEN _____

C.) Does B divide into A evenly?
IF INT(A/B) = A/B THEN _____

In mathematics, the divisibility notion opens up the entire world of

NUMBER THEORY. Evens and odds, factors and multiples, primes and composites -- are all consequences of GOZINTA. And divisibility is an examples of the power of the INT function. Euclid would be so proud of you!

EXERCISES 4.2

1. Translate each of the following sentences into an Applesoft statement.
A.) If X is a factor of B, go to line 50
B.) If C is a multiple of 6, go to line 100
C.) If Q is even go to line 80
D.) If Y is divisible by 2 or 3, go to line 10

2. Divisibility rules for large numbers can be derived from rules for their factors. For instance, we can tell if a number is divisible by 18 by using divisibility rules for its factors 9 and 2. If a number is divisible by 9 and is even, then it is divisible by 18. Use this idea to develop a rule for 36. Then write a program using this rule to check an inputted value to see if it is or is not divisible by 36.

3. Some math students dread the long division process. But the Apple handles it with ease, when we teach him to do it once. Trace the following program.

```
10 READ N
20 DATA 7,13,18
30 LET Q = INT(N/3)
40 LET R = N - 3 * Q
50 PRINT TAB(7);Q;SPC(3);"R.";R
60 PRINT TAB(5); "_______"
65 PRINT
70 PRINT TAB(3);"3)";SPC(1);N
80 GO TO 10
```

4. An improper fraction is one in which the numerator is greater than or equal to the denominator. For example, 3/2, 18/4, 25/5, are improper fraction. Write a program to express each as a mixed numeral. Be sure to express whole numbers without a fractional part. The output should look like:

```
3/2 = 1 1/2
25/5 = 5
```

5. In Section 4.1, you traced a program which in part determines the digits of a three digit number. Use that program to help you design a solution here. A number is divisible by three if the sum of the digits is a multiple of three. Write a program to input a three digit number, add up the digits and check to see if the number is divisible by 3.

SECTION 4.3 STROLL THROUGH NUMBER THEORY

The ability of the Apple to handle topics in number theory depends on the INT function. The early Greeks, some 2000 years ago, did a comprehensive study on the structure of our counting numbers. The INT function allows us to program their work.

Euclid was fascinated, even mystified, with the patterns within the natural numbers. Numbers were identified by sexes. Marraiges of numbers and even bachelorhood had correspondences within the set. One of the most unusual groupings of the numbers was as PERFECT, DEFICIENT, or ABUNDANT. A perfect number is one which is equal to the sum of its proper factors. A number was deficient or abundant depending on whether the sum was respectively less than or greater than the number itself. Write a program to input a number and describe it as perfect, deficient, or abundant.

This being the most difficult problem we've faced up to now, let's see if we can talk through the solution before we tackle a flowchart or program. This problem is really two separate programs. The first task is to find the set of proper factors of the input, N, and to accumulate them into a sum. The second task is to compare N with this sum, S, and branch to the appropriate output.

Let's begin by tackling the factor portion. Just as a little review, take time to write out the set of factors for each of the following:

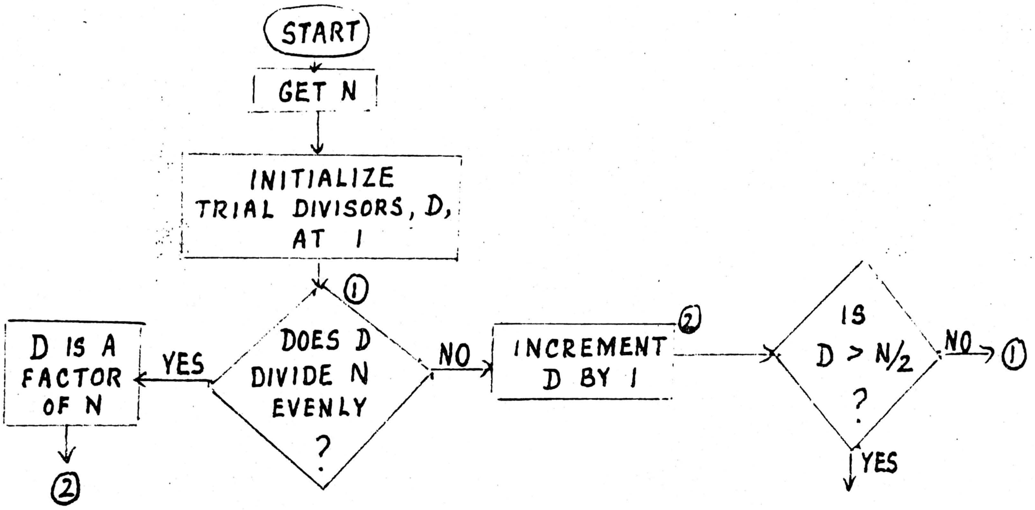
A.) 24
B.) 33
C.) 28
D.) 41

ANSWERS:A.) 1, 2, 3, 4, 6, 8, 12, 24
B.) 1, 3, 11, 33
C.) 1, 2, 4, 7, 14, 28
D.) 1, 41

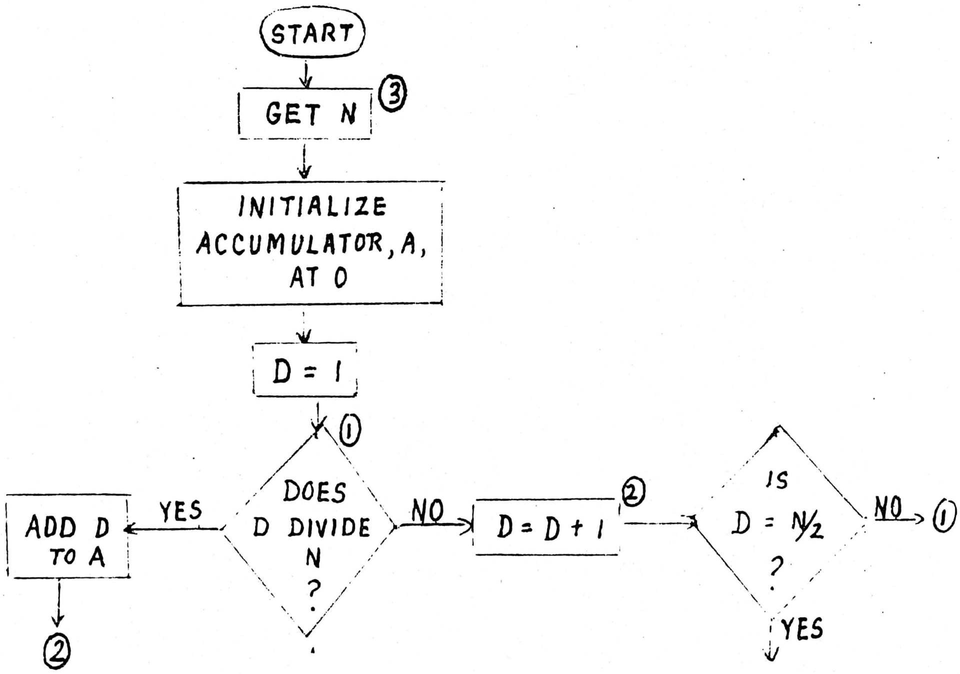
Proper factors are all factors less than the number itself. To get a feel for accumulating the total for each, find the sum of the proper factors. and describe it as perfect, abundant, or deficient.

A.) 24: 36 ABUNDANT
B.) 33: 15 DEFICIENT
C.) 28: 28 PERFECT
D.) 41: 1 DEFICIENT

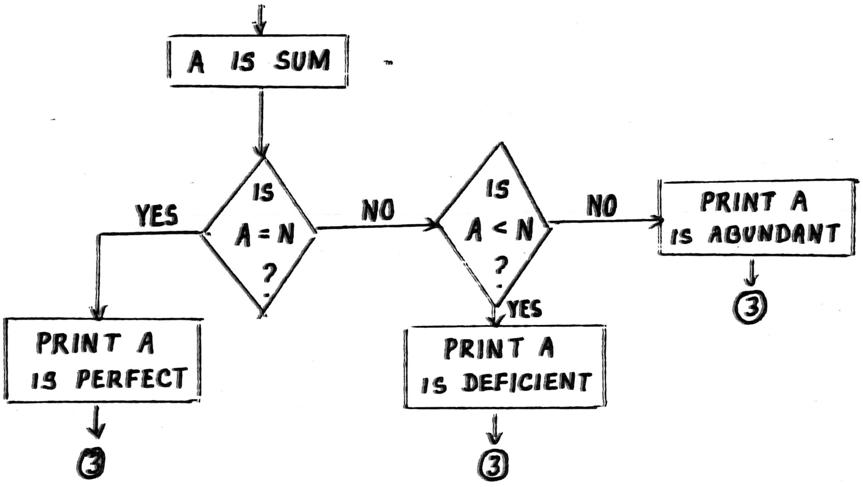
How can we get the Apple to select factors of N? We can do much as you would. We can start with a trial divisor (factor) -- 1. Is 1 a factor of the N? Does it divide evenly into N? If yes, then 1 is a factor. If not, we proceed to a new divisor. We increment the divisor and repeat the procedure. Does 2 divide into N evenly? Does 3,4,5, etc.? But when can we stop? At what point will er be assured of having found the largest proper factor? For 24, do we need to look above 12? If you answered yes, I hope you are ready to explain. We do not need to try divisors greater than half the number. Once we have found the factor representing the halfway position, all succeeding factors will already have been listed. The number in the halfway position will be our test for the trial divisors. The flowchart portion for generating factors might look like this:


START
GET N
INITIALIZE TRIAL DIVISORS, D, AT 1
①
DOES D DIVIDE N EVENLY ?
YES
D IS A FACTOR OF N
②
NO
INCREMENT D BY 1
②
IS D > N/2 ?
NO
①
YES

This flowchart produces a list of proper factors for any given number. Of course, we do not need to output the list. Instead, we wish to accumulate a total for these factors. Here we can use our ACCUMULATOR. This means initializing this total at 0, and incrementing each time a factor is found. These two statements can be inserted into the flowchart, completing the first portion of our problem.


START
GET N
③
INITIALIZE ACCUMULATOR, A, AT 0
D = 1
①
DOES D DIVIDE N ?
YES
ADD D TO A
②
NO
D = D + 1
②
IS D = N/2 ?
NO
①
YES

The second portion is a branching program. We wish to compare the sum of the factors, D, with the original number N. The flowchart for this looks like:



The connector 2 leads us to "GET N". The two flowcharts can be meshed to form one. A flag to end the program and some thought to output format and you should be in a position to write the program. Be sure to follow the NO path through before translating the YES paths. Do this on your own before checking ours!

```
5 PRINT "PERFECT, ABUNDANT, OR DEFICIENT?"
7 PRINT
10 INPUT "INPUT NUMBER BETWEEN 2 AND 2000
         (NEGATIVE NUMBER TO STOP): ";N
20 IF N < = 0 THEN 200
30 LET A = 0
40 LET D = 1
50 IF INT(N/D) = N/D THEN 90
60 LET D = D + 1
70 IF D > N/2 THEN 110
80 GO TO 50
90 LET A = A + D
100 GO TO 60
110 REM: A NOW IS THE SUM OF FACTORS
120 IF A = N THEN 160
130 IF A < N THEN 180
140 PRINT N " IS ABUNDANT", "SUM IS ";A
150 GO TO 7
160 PRINT N " IS PERFECT", "SUM IS ";A
170 GO TO 7
180 PRINT N " IS DEFICIENT", "SUM IS ";A
190 GO TO 7
200 END
```

A very complex problem has been solved. As the programmer, you should take time to trace through with some simple numbers. Use the three values below as input and see if we have succeeded.

A.) 9
B.) 6
C.) 12

(Answers: 9 IS DEFICIENT, 6 IS PERFECT,
12 IS ABUNDANT)

Despite some sophisticated computer searches, the process of finding perfect numbers is still underway. The largest found so far is 33,550,336. And no one has yet proven that there is a maximum size for a perfect number. That gives you something to dabble with in your spare time!

This process of using a trial divisor can be applied to many of the number theory concepts. For example, to discriminate primes, we want none of the trial divisors to divide evenly into a number. Likewise, to determine the Greatest Common Factor (GCF) of two numbers, we want a trial divisor which divides evenly into both numbers. We leave these for exercises, but we hope you agree with us that the INT function does OPEN NEW DOORS.

EXERCISES 4.3

1. Below is the program for generating the entire set of factors for a given number. This has been adapted from our perfect number program. Make any necessary modifications to have the program count the number of factors.

```
10 INPUT N
20 PRINT "DIVISORS OF "N" ARE: ";
30 FOR D = 1 TO INT(N/2)
40 IF INT(N/D) = N/D THEN 70
50 NEXT D
60 PRINT N
65 GO TO 10
70 PRINT D;
80 GO TO 50
```

A prime number can be defined as a natural number which has exactly two divisors. Using this definition, continue to change the program in order to determine whether a given number is prime or composite.

2. As mentioned in our discussion, another approach to finding prime numbers is to establish a set of trial divisors and test to see if any of them is a factor of the given number. Write a program which generates the numbers from 2 to 100 and outputs a list of prime numbers. Output the list in column form as shown below:

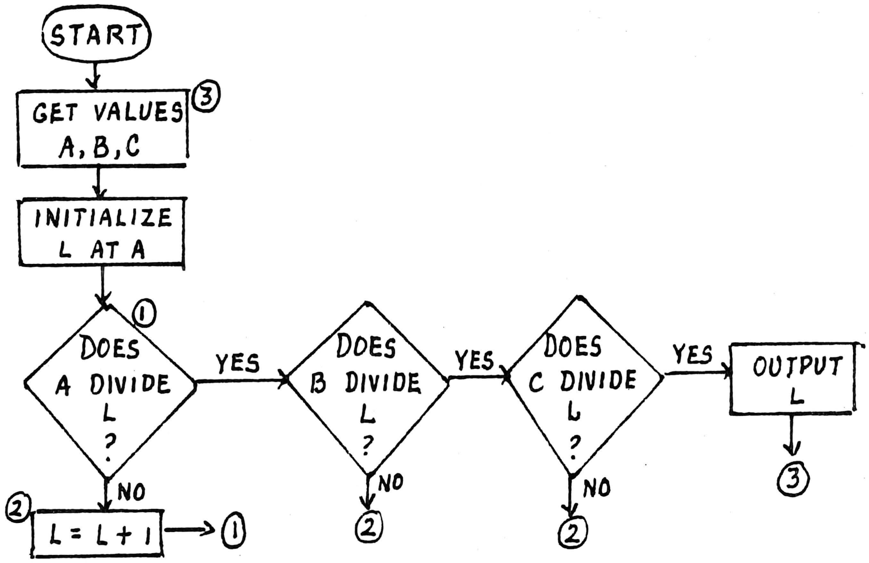
```
2              3              5
7              11             13
```

3. Trace the following program carefully. After completing it, briefly describe the function of the program.

```
10 READ N
20 DATA 24,15,19
25 PRINT N; TAB(10);
30 LET T = 2
40 IF T > N THEN 10
50 IF INT(N/T) = N/T THEN 70
60 LET T = T + 1
70 GO TO 40
80 PRINT T; SPC(2);
90 LET N = N/T
100 GO TO 40
110 END
```

4. The Greatest Common Factor of two numbers is the largest number which is a factor of each. The GCF of 20 and 35 is 5. Write a program to accept two numbers and determine their GCF. This looping program must test divisors until the divisor is a factor of both numbers.
(HINT: Make the loop step by -1 starting at the smallest number.)

5. Translate the following flowchart program to an Applesoft program. Provide three sets of data and trace through your program. Do you recognize the function of this program?


START
GET VALUES A,B,C
③
INITIALIZE L AT A
①
DOES A DIVIDE L ?
YES
DOES B DIVIDE L ?
YES
DOES C DIVIDE L ?
YES
OUTPUT L
NO
②
L = L + 1
①
NO
②
NO
②
③

AT THE KEYBOARD SECTION 4

PEEKING INTO THE FUTURE

With more advanced programming, you will see new and different commands. This section is not intended to provide an in-depth discussion of each statement, but to give you some exposure to them. Our approach will be to present a program that incorporates several of these commands, and then discuss their function within the program.

Our discussion program below is designed to do several operations. In one of the earlier Sections, we discussed how the PRINT statement can be altered to have an output that is flashing or in inverse color. The importance of these commands is their ability to draw attention to the screen. The flash command, however, is sometimes too flashy. The inverse/normal/inverse/normal output can be distracting, especially when there is other text on the screen. One program feature converts this inverse/normal output to a normal/nothing/normal/nothing output pattern. A second feature of this program is a mystery key finder. When a key is pressed, the program is designed to determine which key was actually typed, and a salutation of thanks for pressing that specific key is displayed.

Here is the program:

```
10 HOME
15 LET QS = CHRS(162)
20 VTAB 9
21 HTAB 1
30 PRINT "PRESS     KEY."
40 FOR X = 1 TO 200
42 NEXT X
50 VTAB 9
51 HTAB 1
60 PRINT "PRESS ANY KEY."
70 FOR X = 1 TO 200
72 NEXT X
80 LET K = PEEK (-16384)
85 IF K < 161 THEN 20
90 POKE -16368,0
95 VTAB 9
96 HTAB 1
100 PRINT "THANKS FOR PRESSING THE ";QS;CHRS(K);"KEY."
```

The first command, HOME, clears the screen in preparation for the PRINT statements that follow. The last NEXT command assigns QS the value of another string in memory, the character string number 162, abbreviated CHRS(162). As with the COLOR command in graphics, each key on the keyboard has a numerical value associated with it. When you type an "A", for instance, it is converted into a code, called ASCII (pronounced ASK E). For the letter A, the ASCII code is 193; the letter B 194; the letter C 195, and so on. The CHRS function outputs the string or character which corresponds to the argument value. As proof, you could type...

```
10 PRINT CHRS(193)
```

...and the letter A would be displayed.

Returning to our program, 162 is the ASCII code for quotes.

```
PRINT CHRS(162)
```

This would output one set of quotes. This brings up an interesting point. Up to this time, we have been unable to produce an output that contains quotes.

```
10 PRINT " 'HELLO'"
```

The above would generate an output of: 'HELLO' but this may not be as desirable as "HELLO".

```
10 PRINT " "HELLO" "
```

Because the first character in the above line in quotes is a space, the output 0. The word HELLO is not in quotes, so the computer checks its memory to see if there is a variable addressed HELLO. There is no such address so the output is the default value of zero. The last set of quotes, like the first, contains a space.

With the use of ASCII codes, we are able to generate the output "HELLO" by using the line:

```
10 PRINT CHRS(162);"HELLO";CHRS(162)
```

The output is character string 162, a set of quotes. Then the word HELLO and another set of quotes appear. We will use the quotes later in this program.

Once again, back to the matter at hand. Lines 20 and 21 are used for formatting the PRINT statements which follow -- TAB 9 down and 1 over. Line 30 prints the phrase: PRESS (five spaces) KEY. The reason that the spaces are contained in this line is not yet evident, but hang on! Line 40 is a delay routine with the NEXT command as the following statement. Once again, lines 50 and 51 use the same format as lines 20 and 21 so that the next PRINT statement will be placed in the exact location as the first. Here is where the key to the program lies. Line 30 gives the output:

```
PRESS     KEY.
```

Normally, the output of line 60 would be placed a line down from any previous PRINT statement, but because of the VTAB, HTAB statement immediately preceding line 60, the output is in the same location:

```
PRESS ANY KEY.
```

We have created one flash. Originally, the word ANY was not on the screen, now it is. The delay routine in line 70 slows down this flash process.

Now for the fun part! PEEK and POKE are two commands that have an invaluable function in advanced programming, as you will see. We have established that the memory banks of the computer are comprised of many locations that can be addressed and filled. Some of these locations are for numerical variables, some are for strings, and some serve other functions. One of these is to determine if any key on the keyboard has been pressed. In simplified terms, if a mail slot, specifically number -16384, is filled, a character has been typed. If this mail slot is empty, the keyboard has not been touched. The PEEK command is the statement we use to peer into a memory location. If the user pressed the "A" key, for example, the variable K has a value of 193; the "B" key gives it 194; the "Z" key 218. Remember, any key has an ASCII value associated with it. The PEEK command assigns K this value.

The range of ASCII values printed in the normal mode is from 161 (the "!" key) to 218 (the "Z" key). Consequently, K must have a value that falls in this range of numbers. A K value of less than 161 can be due to only one circumstance -- the keyboard has not been touched. This is the function of line 85. If a character has not been typed, return to line 20 and continue our flash process. If a key has been pressed, proceed to line 90.

The function of the command POKE, -16368,0 in line 90, is to reset memory location -16384. If this portion of the program is encountered at a later time, the mail slot will be empty. Just as the INPUT and LET commands placed a value into an assigned location, the POKE command inserts a value into a specific location. In this case, the command places zero in memory location -16384 and in doing so, erases all stored values. Note that the POKE and PEEK commands have different arguments, but strangely both refer to the same location. Lines 95 and 96 prepare for the next PRINT output to be placed in the same location as any succeeding PRINT output.

The program ends with considerable flair. The first portion of the output in line 100 is obvious.

```
THANKS FOR PRESSING THE
```

Referring to line 15, QS represents a set of quotes. We now have:

```
THANKS FOR PRESSING THE "
```

The CHRS function uses the ASCII value of a key as its argument. The variable K has a value equal to the ASCII value of the key that was pressed. Therefore, line 100 adds to THANKS FOR PRESSING " the letter that was typed. Here K = 193.

```
THANKS FOR PRESSING THE "A
```

Another set of quotes follows and the word KEY. Well, there it is folks!

```
THANKS FOR PRESSING THE "A" KEY.
```

Take time to explore the use of the CHRS function and the use of the PEEK and POKE commands. A reference source for ASCII values and for specific memory location to use with PEEK and POKE is essential to have. good luck with future discoveries!

SECTION 4.4 A PROGRAM PER CHANCE

A most interesting application of the Apple is in the simulation of events which involve "chance". Whether it is rolling dice, guessing high-low, or generating drill problems, the computer can be applied. We are going to see how these problems in chance are solved.

Create Apple the coin tosser. Simulate a probability experiment of flipping two coins. Allow for input of the number of flips and calculate the actual probability of the three possible outcomes -- two heads, two tails, or one head and one tail.

To the Apple, creating the notion of chance means randomness. It's picking numbers out of a hat. And to do this the Apple has what is called a <u>RANDOM</u> <u>NUMBER</u> <u>GENERATOR</u> function, <u>RND</u>. The RND function simply provides us numbers randomly chosen from a huge list of numbers. The numbers are between 0 and 1; actually .00000000 < RND < .999999999. Zero is on the list, 1 is not. Imagine if you will, numbering 1,000,000,000 chips each with one of these numbers. Using the RND function selects one chip and enables us to use the number on it. And each time we ask for another number, the chip is replaced and the drawing process is repeated. The argument for this function is unlike other functions. It is a dummy argument; it has no effect on the numbers chosen as long as it is a positive integer. Using RND(1) or RND(3.45) does not change the procedure. The following program generates five random numbers.

```
10 FOR X = 1 TO 5
20 PRINT RND(1)
30 NEXT X

.268001130
.419217095
.878831482
.123353160
.368373372
```

You will notice the nine-place decimals. It isn't always desirable to work with such wierd decimals between 0 and 1. It is possible to control the kinds of numbers in the list through the use of INT and our basic math operations.

For example, suppose we need to pick numbers from 0 to 9. How could we alter our random numbers? It is possible to control the range of numbers by use of a factor. If we multiply the possible RND values by 10, we generate numbers from 0 to 10 (not including 10). Multiplication by 100 generates numbers from 0 to 100. A factor of 20 and the list is between 0 and 20. Altering line 20 produces a new list of five random numbers.

```
20 PRINT RND(1) * 10

2.57847407
9.29521128
3.69109253
.798985089
6.08766574
```

We have rational numbers greater than or equal to 0 and less than 10.

Applying the INT function to this random number will chop off the unwanted fraction producing the integers 0 to 9.

```
20 PRINT INT(RND(1) * 10)
```

```
1
0
5
7
9
```

Along with the INT function, the factor indicates the number of possible outcomes. Multiplying by 10 produces ten random numbers from 0 to 9. For a game involving dice, another choice of numbers is called for -- 1 to 6. We can use the same techniques. There are six possible integer outcomes, and therefore we want to multiply by a factor of 6.

```
20 PRINT INT(RND(1) * 6)
```

However, this produces random integers from 0 to 5. What we need to do is to shift the entire range by 1. To do this we will use an addend of 1.

```
20 PRINT INT(RND(1) * 6) + 1
```

And here are five random numbers generated.

```
2
3
6
4
3
```

In summary, the INT function will allow us to generate integer rather than decimal random numbers. To control the number of possible outcomes, RND is multiplied by that number. To control the start of the random sequence, an addend is needed. For each of the following write the PRINT statement to generate a random X in the indicated range.

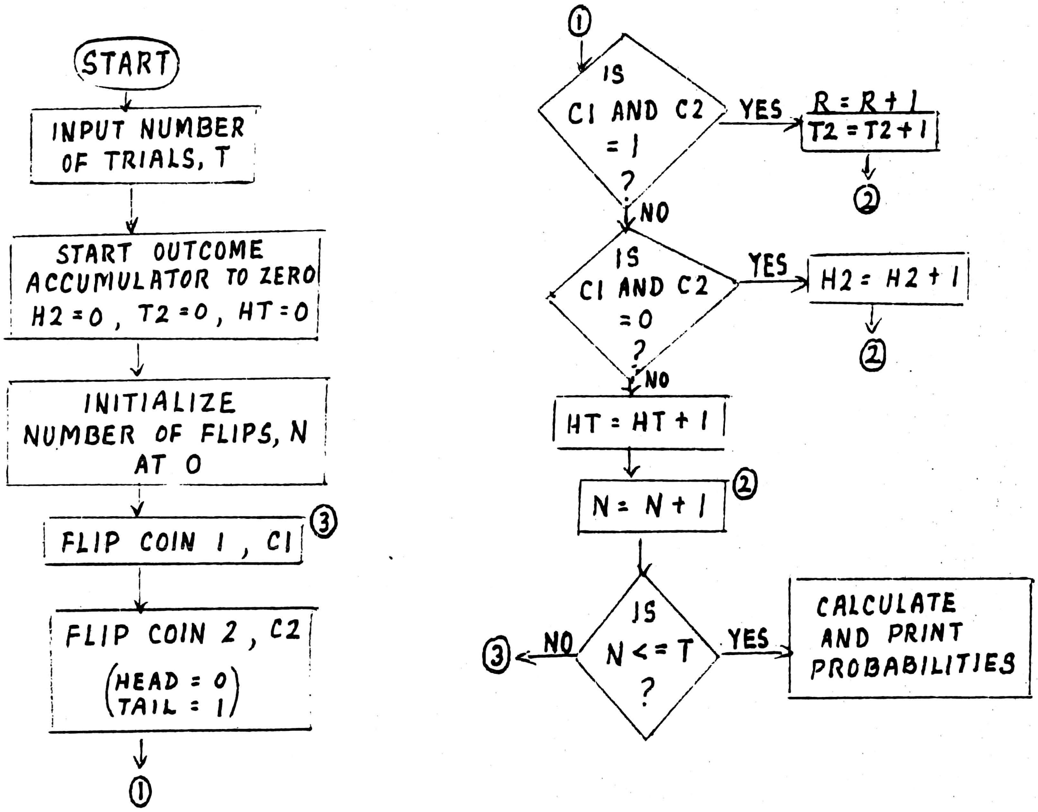
A.) $0 < X < 15$, X IS A RATIONAL
B.) $0 < X < 100$, X IS AN INTEGER
C.) $1 < X < 100$, X IS AN INTEGER
D.) $50 < X < 75$, X IS AN INTEGER
E.) $-10 < X < -5$, X IS AN INTEGER

```
ANSWERS A.)  RND(1) * 15
        B.)  INT(RND(1) * 101
        C.)  INT(RND(1) * 100) + 1
        D.)  INT(RND(1) * 26) + 50
        E.)  INT(RND(1) * 6) - 10
```

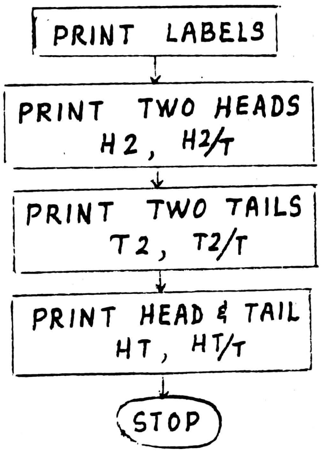
Back to our coin tossing problem. To simulate the two possible outcomes of the flipping, we will need to generate two random numbers. We can let 0 represent a HEAD and 1 represent a TAIL. What will the statement to generate these two random integers look like?

```
INT(RND(1) * 2)
```

We need to create a loop to flip two coins, each another random number, and to record the results. To record, we need to set up a separate counter for each of the three outcomes. Once the loop is finished, we can use the counters to compute the probabilities. PROBABILITY OF EVENT = NUMBER OF EVENT OCCURRENCES/NUMBER OF TRIALS. A flowchart for the coin tossing loop might look like this:


START
INPUT NUMBER OF TRIALS, T
START OUTCOME ACCUMULATOR TO ZERO H2 = 0, T2 = 0, HT = 0
INITIALIZE NUMBER OF FLIPS, N AT 0
FLIP COIN 1, C1
③
FLIP COIN 2, C2 (HEAD = 0 TAIL = 1)
①
①
IS C1 AND C2 = 1 ?
YES
R = R + 1
T2 = T2 + 1
②
NO
IS C1 AND C2 = 0 ?
YES
H2 = H2 + 1
②
NO
HT = HT + 1
N = N + 1
②
IS N <= T ?
NO
③
YES
CALCULATE AND PRINT PROBABILITIES

The probability of each outcome can be computed by dividing the frequency -- T2, H2, and HT -- by the number of trials, T.


PRINT LABELS
PRINT TWO HEADS H2, H2/T
PRINT TWO TAILS T2, T2/T
PRINT HEAD & TAIL HT, HT/T
STOP

Try your hand at translating this into a program. Be sure to follow the NO path before going to the YES path. We have also had the frequency of each outcome printed out, as well as the probability.

```
5 PRINT "COIN FLIPPING EXPERIMENT"
10 INPUT "NUMBER OF FLIPS? ";T
20 LET H2 = 0
30 LET T2 = 0
40 LET HT = 0
50 FOR N = 1 TO T
60 LET C1 = INT(RND(1) * 2)
70 LET C2 = INT(RND(1) * 2)
80 IF C1 = 1 AND C2 = 1 THEN 120
90 IF C1 = 0 AND C2 = 0 THEN 140
100 HT = HT + 1
110 GO TO 150
120 T2 = T2 + 1
130 GO TO 150
140 H2 = H2 + 1
150 NEXT N
160 PRINT TAB(20);"FREQUENCY"; TAB(40);"PROBABILITY"
170 PRINT "TWO HEADS"; TAB(25); H2; TB(45); H2/T
180 PRINT "TWO TAILS"; TAB(25); H2; TAB(45); T2/T
190 PRINT "HEAD & TAIL"; TAB(25); HT; TAB(45); HT
200 END
```

We ran this program three times; the output is listed below. Each time the results are different -- but we would expect the RND function to produce this.

```
  FREQUENCY                PROBABILITY
TWO HEADS                    27                    .27
TWO TAILS                    24                    .24
HEAD & TAIL                  49                    .49

TWO HEADS                    26                    .26
TWO TAILS                    32                    .32
HEAD & TAIL                  42                    .42

TWO HEADS                    237                   .237
TWO TAILS                    243                   .243
HEAD & TAIL                  520                   .52
```

In this last execution, with 1000 trials, you can see that the actual probabilities are approaching the theoretical probabilities of .25, .25, and .5. And you can see the potential of the computer in such experiments. The Apple flips 1000 times in 15 seconds, 10000 times in less than 3 minutes. The power of repetition is obvious. We could hardly keep our heads up for that many flips!

EXERCISES 4.4

1. Write the Applesoft statement to print a random integer, X, between the integers given. The integers should also be printed out.

A.) 0 and 5
B.) 1 and 5
C.) 20 and 40
D.) -10 and 0
E.) -15 and -5
F.) A and B

2. Write a program to generate random numbers from 0 to 1 (not including 1). The program, however, should print out only those which are greater than .5. What experiment could we simulate by using this random number test?

3. Write an assignment statement which will choose an number in the appropriate set for each of these experiments.

A.) Rolling one die from a pair.
B.) Select a card from a deck of cards
C.) Spin the roulette wheel
D.) Choose a four digit winner in the lottery.

4. Conduct a programming experiment to roll two dice one hundred times. During the experiment count the number of times the sum of the numbers on the two dice is 7. Output the total count and the total number of rolls. Can you see why seven plays an important role in the game of CRAPS!

5. 6 + 5 + 10 + 5 + 2 represents the sum of five random numbers selected from 1 to 10. Write a looping program to find the sum of five randomly selected integers from 1 to 10. Then modify it into a nested loop by carrying out this process fifteen times. The output sould look like this:

```
 NUMBERS         SUM
1 2 6 10 2        21
3 3 3  9 7        26
```

SECTION 4.5 GAMING RANDOMLY

Since so many games involve chance, the RND function is a part of most every computer game program. Rolling dice for "monopoly" or dealing cards for "Old Maids", luck is there. And in a program, so is the RND function.

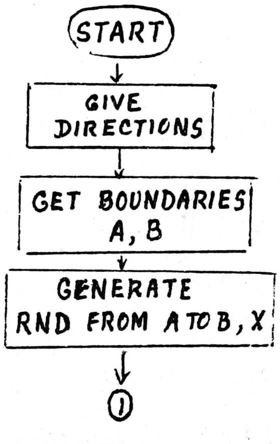
```
253 IS TOO LOW
THAT'S YOUR THIRD GUESS
GUESS AGAIN!
```

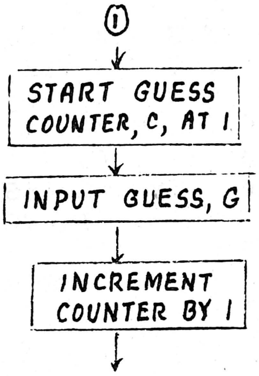
That's the Apple giving directions to a HIGH-LOW game. Write a program to have the Apple pick a hidden number. Then allow the player to guess the number, by using computer given clues of "HIGHER" or "LOWER". Be sure to count the guesses used.

As you read this programming project, key words should help ideas materialize -- "pick a hidden number" means the RND function; "clues of HIGHER or LOWER" indicates the need for branching; "player to guess" translates as input; "count the guesses" establishes the need for a counter. With all of these clues, writing a flowchart should be "a piece of cake".

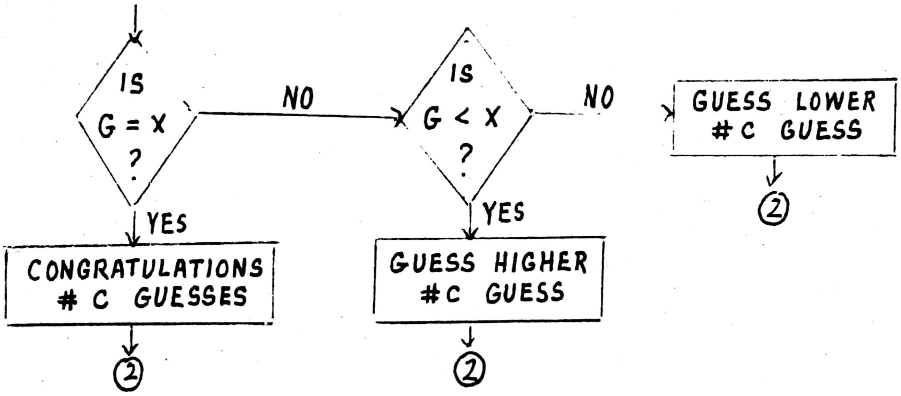
Having the Apple pick a hidden number is easy now that we have our RND function. The only question in having th Apple pick a hidden number is the range of the possible numbers. To spice up the challenge, let's allow the player to choose the beginning and end of this range. Inputting 100,000, for example, would mean choosing a number between 100 and 1000. While we are constructing the flowchart, think about what the RND statement will look like to use A and B to produce the correct list of possible numbers.

If we are going to count the guesses, the counter needs to be initialized prior to asking for the play's first guess. Our flowchart beginning might look like this:





Once the guess is inputted, our branching portion begins. We need to compare G with X. If G = x, the player has succeeded. With G < X, a "HIGHER" clue should be given; with G > X, a "LOWER" guess is called for. Under either of these circumstances, another guess should be requested.



Once the player has guessed the number, the options are open to the programmer. Perhaps the player can be given the option of playing again. And perhaps we can ask if he/she wishes to have directions again or wishes to reset the boundaries.

Our version of the program follows. Look carefully at the options at the end of the program. Two different executions of the program follow.

```
10 PRINT TAB(14); "HIGHER-LOWER"
11 PRINT
12 PRINT
15 PRINT "THIS IS THE NUMBER GUESSING GAME OF"
16 PRINT "HIGHER-LOWER.  APPLE WILL SELECT"
17 PRINT "A SECRET NUMBER.  YOUR JOB IS TO GUESS"
18 PRINT "THE NUMBER USING THE APPLE'S CLUES"
19 PRINT
20 PRINT "INPUT THE BOUNDARIES FOR APPLE'S"
30 INPUT "RANDOM PICKS. (SMALLEST, LARGEST): ";A;B
40 LET X = (INT(B - A + 1) * RND(1)) + A
42 LET C = 0
45 PRINT
50 INPUT "YOUR GUESS: ";G
55 PRINT
60 LET C = C + 1
70 IF G = X THEN 150
80 IF G < X THEN 130
90 PRINT "GUESS # ";C;" -- ";G
100 PRINT "IS TOO HIGH.  GUESS LOWER"
110 GO TO 45
120 PRINT "GUESS # ";C;" -- ";G
130 PRINT "IS TOO LOW.  GUESS HIGHER"
140 GO TO 45
150 PRINT "CONGRATULATIONS! ";G;" IS CORRECT"
160 PRINT "IT TOOK ";C;" GUESSES."
165 PRINT
170 INPUT "DO YOU WISH TO PLAY AGAIN? (YES OR NO) ";AS
180 IF AS = "NO" THEN 999
190 INPUT "DO YOU NEED DIRECTIONS AGAIN? ";BS
200 IF BS = "YES" THEN 12
210 INPUT "DO YOU WISH TO RESET BOUNDARIES? ";CS
220 IF C$ = "YES" THEN 19
230 PRINT "THE APPLE HAS ANOTHER NUMBER"
240 GO TO 40
999 END
```

```
]RUN
              HIGHER-LOWER


THIS IS THE NUMBER GUESSING GAME OF
HIGHER-LOWER.  APPLE WILL SELECT
A SECRET NUMBER.  YOUR JOB IS TO GUESS
THE NUMBER USING THE APPLE'S CLUES

INPUT THE BOUNDARIES FOR APPLE'S
RANDOM PICKS. (SMALLEST,LARGEST): 1,100

YOUR GUESS: 50

GUESS #1 -- 50
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 75

GUESS #2 -- 75
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 87

GUESS # 3 -- 87
IS TOO HIGH.  GUESS LOWER

YOUR GUESS: 81

GUESS # 4 -- 81
IS TOO HIGH.  GUESS LOWER

YOUR GUESS: 78

GUESS #5 -- 78
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 76

GUESS #6 -- 76
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 80

CONGRATULATIONS! 80 IS CORRECT
IT TOOK 7 GUESSES.

DO YOU WISH TO PLAY AGAIN? YES
DO YOU NEED DIRECTIONS AGAIN? NO
DO YOU WISH TO RESET BOUNDARIES? NO
THE APPLE HAS ANOTHER NUMBER
```

```
             HIGHER-LOWER


THIS IS THE NUMBER GUESSING GAME OF
HIGHER-LOWER.  APPLE WILL SELECT
A SECRET NUMBER.  YOUR JOB IS TO GUESS
THE NUMBER USING THE APPLE'S CLUES

INPUT THE BOUNDARIES FOR APPLE'S
RANDOM PICKS. (SMALLEST,LARGEST): 1,20

YOUR GUESS: 10

GUESS #1 -- 10
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 15

CONGRATULATIONS! 15 IS CORRECT
IT TOOK 2 GUESSES.

DO YOU WISH TO PLAY AGAIN? YES
DO YOU NEED DIRECTIONS AGAIN? NO
DO YOU WISH TO RESET BOUNDARIES? YES

INPUT THE BOUNDARIES FOR APPLE'S
RANDOM PICKS. (SMALLEST,LARGEST): 1,10

YOUR GUESS: 5

GUESS #1 -- 5
IS TOO LOW.  GUESS HIGHER

YOUR GUESS: 8

CONGRATULATIONS! 8 IS CORRECT
IT TOOK 2 GUESSES.

DO YOU WISH TO PLAY AGAIN? NO
```

For all of us who enjoy games and game playing, the RND function provides us with LUCK. So although we cannot control our own fate, we can have the Apple simulate it. Enjoy these new functions to produce your own simulations and games. It's your TURN!

EXERCISES 4.5

1. The role of dice in games was given a shot in the arm by the new game of "DUNGEONS AND DRAGONS". In this game there are two different dice -- each with a different number of faces. The two dice are examples of the five PLATONIC SOLIDS. Write a statement to store in R, a random number that might be generated on each die.

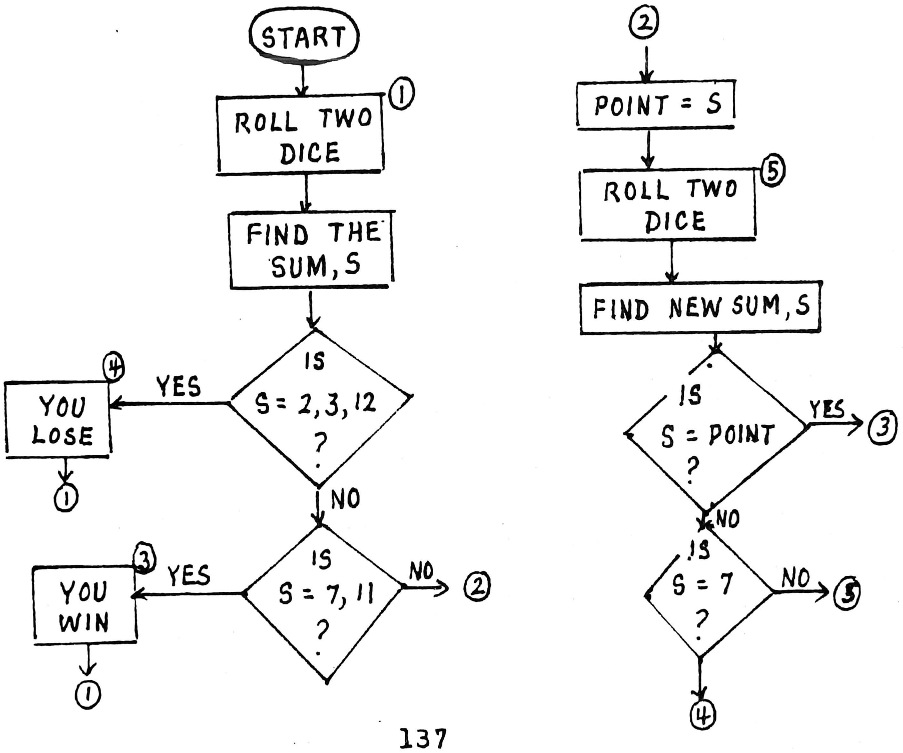
A.) Tetrahedron
B.) Cube
C.) Octahedron
D.) Dodecahedron
E.) Icosahedron

You might have to consult a D & D wizard to find the number of faces on each die.

2. Do you need a little practice with your multiplication facts. Provide a little quiz by writing a program which generates two random numbers. Select the numbers at random from the integers from 1 to 12. Output the numbers as a product question. Then input the answer and check to see if the answer is correct or not.

3. Write a program to conduct a probability experiment. The program will determine the probability of a number between 1 and 10 being divisible by 3. Generate one hundred random integers in the correct range. For each check to see if they are divisible by 3. If yes, maintain a counter to be incremented. If not, simply get another number. The outcome of the experiment should be the number of times out of one hundred that the number was divisible by 3.

4. Craps is an ancient game of dice rolling. Below is a flowchart containing the rules. Translate it into Applesoft. Add dialogue to improve the program.



5. You're late for school. But you need some change for all those things that surround school life: bus fare, lunch, soda, candy. You reach in the CHANGE BOWL and grab a handful. You're off. But how much money did you actually pick up? Determine randomly the number of coins you selected. In the bowl were five nickels, four dimes, three quarters, and fourteen pennies. The program should calculate the total amount of money you grabbed from the bowl.

A LITTLE BIT MORE SECTION 4

ECONOMY SIZE

We would be wrong to close our discussion of abbreviations and short cuts without mentioning a very special capability of the Apple. Applesoft allows a series of statements to be entered on a single line if they are separated by colons. In effect we can pack together several statements into one. Presented below are two programs that do the identical task.

```
10 LET A = 5
20 LET B = 10
30 LET Q = 2
40 PRINT (A + B) * Q
```

```
10 LET A = 5:LET B = 10:LET Q = 2:PRINT
  (A + B) * Q
```

We call line 10 a <u>multiple</u> <u>statement</u> <u>line</u>. Note how the colon separates each command. This second program has been condensed into one line. You might wish to write out your program without multiple statement lines to prevent confusion. Later the program can be rewritten in condensed form. There are two reasons to use multiple statements. First, they save typing time. Secondly, they conserve space in the Apple's memory. However, unless your program is an extremely complex, long one, saved space is really inconsequential.

No shortcut goes without warnings. And indeed we need to use caution when using this shortcut. Several examples will help point out some restrictions.

```
10 PRINT "HELLO": GO TO 20: PRINT "HOW ARE YOU TODAY"
20 INPUT AS
30 IF AS = "GOOD" THEN 60
40 PRINT "TOO BAD"
50 GO TO 70
60 PRINT "SO AM I"
70 END
```

We have used multiple statements in line 10. But if you look closely at the line, the question "HOW ARE YOU TODAY" is never encountered. The preceding GO TO statement transfers us before reaching the final PRINT statement in this line. So BEWARE -- a GO TO may only be used as the last statement in a multiple line, not in the middle somewhere. The same is true for a REM statement. Once the Apple encounters a REM command, the rest of the line is ignored -- even another command. The REM statement must come at the end of a multiple statement line.

Another problem can arise with transfer statements. Look at the two programs below:

```
10 PRINT "NUMBER GUESS"
20 INPUT "INPUT NUMBER?";N
30 IF X <> 15 THEN 20
40 PRINT "CORRECT"
```

```
10 PRINT "NUMBER GUESS": INPUT "INPUT NUMBER:";
   N : IF X <> 15 THEN 20
40 PRINT "CORRECT"
```

The second example is an attempt to condense the program. A problem arises for this student when he tried to transfer to the INPUT statement. Since INPUT is buried in the middle of a multiple statement line there is no way to transfer to it, without repeating the entire line. So again, BEWARE, in order to transfer to a statement in a multiple statement line, it must be the first one. Don't make a statement inaccessible. DATA statements are a special case of this rule. In order for the Apple to find the data without having to execute other statements, the DATA statement may never be used in a multiple statement line. DATA must always stand alone.

A final example demonstrates another problem. The use of the colon may effect the output in unusual ways if a conditional transfer is used.

```
10 PRINT "DO YOU WANT A MATH PROBLEM?":
   INPUT AS: IF AS = "YES" THEN 20: GO TO 999
20 PRINT "WHAT IS 25/3.14?"
30 INPUT X: IF X = 25/3.14 THEN 60
40 PRINT "NO, THE ANSWER IS"; 25/3.14
50 GO TO 999
60 PRINT "GOOD"
70 END
```

In this program, line 10 asks if you want a math problem and then waits for and tests the input. If the input is "YES" the program transfers to line 20. If the input is any other word, and consequently does not pass the test, the computer moves to the next statement number. WHOOPS! Not GO TO 999, but to line 20. The statement GO TO 999, which should have been encountered if the input was "NO", is never reached. However in line 30, a multiple statement line has been used correctly. Here, if the test is passed the program transfers to 60. If not, the program moves to the next line, giving the correct answer. So beware -- use the IF-THEN statement only at the end of a multiple statement line to prevent transfer mistakes.

Caution needs to be used. But don't let that discourage you from using multiple statements. As a final example let's stretch its use to the limit correctly.

```
10 PRINT "THIS PROGRAM WILL ADD ANY TWO NUMBERS"
20 PRINT "THAT YOU INPUT"
30 FOR X = 1 TO 500
40 NEXT X
50 INPUT "WHAT IS THE FIRST NUMBER?";X
60 INPUT "WHAT IS THE SECOND NUMBER?";Y
70 PRINT "THE SUM OF ";X;" AND ";Y; " IS ";X + Y
```

```
10 PRINT "THIS PROGRAM WILL ADD ANY TWO NUMBERS:
   PRINT "THAT YOU INPUT": FOR X = 1 TO 500:
   NEXT X: INPUT "WHAT IS THE FIRST NUMBER"; X:
   INPUT "WHAT IS THE SECOND NUMBER"; Y: PRINT
   "THE SUM OF "; X;" AND "; Y; " IS "; X + Y
```

WOW. All of that in one line! Used properly, that's quite an economy measure!

REVIEW EXERCISES

CHAPTER FOUR

1. Translate each of the following statements into acceptable BASIC.
A.) Output N rounded to the nearest tenth.
B.) Is X divisible by 8?
C.) Store in X a random integer from 0 to 50?
D.) Is D an even number?
E.) Print a random integer from 20 to 30.
F.) Round the money M, to the nearest dollar.
G.) In Y store a random number from -20 to 0 (not including 0).
H.) Is C a multiple of B?

2. Write a program to check if a number is divisible by 25, by checking to see if it is divisible by 5 twice.

3. Trace the following program with a table of STORED VALUES and the OUTPUT.

```
10 LET N = 5
20 LET D = 7
30 PRINT N; " "; D; "=";
40 PRINT ".";
50 FOR P = 1 TO 10
60 LET X = N * 10
70 LET Q = INT(X/D)
80 PRINT Q
90 LET R = X - D * Q
100 LET N = R
110 NEXT T
120 END
```

4. Write a program to reduce a proper fraction to lowest terms. Use the GCF method, where both the numerator and denominator are divided by their GCF. If the fraction is already simplified, output that.

5. Write a program to test your powers of ESP. The computer will pick ten random numbers from 1 to 5. Read into the program the ten numbers in a row you predict the computer will pick. Then have the computer check each pick and count the correct guesses you made. Have the Apple comment the success rate.

8-10 SUPER SOOTHSAYER
5-7 LUCKY GUESSER
1-4 MY DOG CAN DO BETTER

6. Write a program to play the computer in the game ROCK-PAPER-SCISSORS. The player picks one of these and the computer randomly selects one. To determine the winner each time, consult the table below. Keep count of the number of games and the number of wins.

ROCK BREAKS SCISSORS
SCISSORS CUT PAPER
PAPER COVERS ROCK

CHAPTER 5

NEW HORIZONS

Section 5.1 Face To Face With Graphics
Section 5.2 Doing It Up With Lo-Res
At The Keyboard: Paddling Downstream
Section 5.3 Magnifying The Possibilities
Section 5.4 Moving Into Hi-Res
A Little Bit More: The Sound Of Input
Review Exercises

SECTION 5.1 FACE TO FACE WITH GRAPHICS

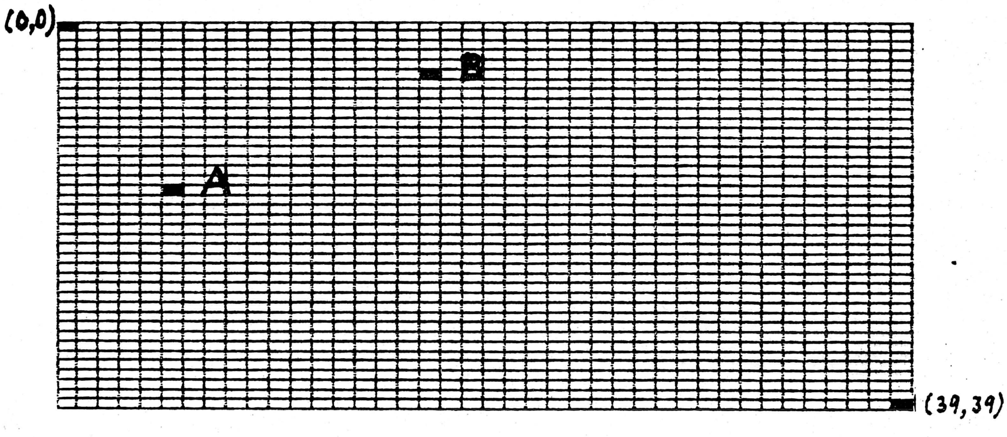
Until now we have concentrated on Applesoft statements in the TEXT mode. We are ready now to add a completely new dimension to our repertoire of skills by exploring the GRAPHICS mode. As in a textbook, the computer has different "pages" devoted to different modes. There are pages for TEXT, written program material. There are also pages for graphics. There are two different graphics pages -- one for LOW RESOLUTION (LO-RES) GRAPHICS and one for HIGH RESOLUTION GRAPHICS (HI-RES).

Here, we are going to investigate the LO-RES page and the statements associated with this form of graphing.

In the 1960's filmed musical "Bye Bye" Birdie, Dick Van Dyke has a musical encounter with a cartooned smiling face. It was soon after that the infamous "HAPPY FACE" came into the commercial world. It appears on letterhead, stickers, pens, and everywhere else imaginable. We would like it to appear one more place -- on the Apple monitor screen. Write a program that will draw a HAPPY FACE.

To begin, we need to familiarize ourselves with the LOW RESOLUTION page. It divides the screen into an invisible 40 X 40 grid system, like that of a sheet of graph paper. There are forty rows and forty columns, subdividing the grid into rectangular-shaped blocks, called bricks. The rows and columns are each labeled from 0 to 39. And each brick can be named using a pair of numbers or coordinates. The first number in the pair corresponds to the column number and the second coordinate corresponds to the row number.

Thus, a brick in the left hand corner is labeled (0,0); the brick in the bottom right corner has coordinates (39,39). Notice that though the last number on the graph is 39, there are indeed 40 columns and rows. The numbering begins at 0. The picture below demonstrates the grid system. The points (0,0) and (39,39) are indicated. The brick labeled A has coordinates (5,18); brick B is (18,5). Remember, the grid is invisible!



The grid occupies the top portion of the screen, while the bottom four lines are reserved for a few lines of programming text and output. In LO-RES graphics, the bricks can be depicted in sixteen different colors, ranging from black to hot pink to white. Each color corresponds to a number, in order to identify it, although the actual hue depends on the monitor being used. To draw in red, we would use the number 1; in brown, number 8 would be used. Below is a table of the colors and their corresponding numbers. If being used with a black and white video display, the colors will be displayed in shades of gray and black.

BLACK	0	BROWN	8
RED	1	ORANGE	9
DARK BLUE	2	GRAY	10
PURPLE	3	PINK	11
DARK GREEN	4	GREEN	12
GRAY	5	YELLOW	13
MEDIUM BLUE	6	AQUA	14
LIGHT BLUE	7	WHITE	15

To take our first steps toward becoming first rate artists, we must indicate to the computer that we intend to use the LO-RES page. The program statement to do this is GR.

```
10 GR
```

The GR statement erases the text page and prepares the Apple to accept LO-RES statements. It must be the first statement in a graphics program. In order to leave either of the graphics modes and return to the normal text programming mode, the TEXT statement is used. This simply drops the LO-RES page. When a TEXT statement is executed, the graphs currently on the page are lost and the screen is filled with "garbage" characters. This garbage will disappear as the text scrolls upward or can be removed by using the HOME command.

Prior to drawing points and lines, we must specify the color to be used. The COLOR variable is assigned a value.

```
20 COLOR = 13
```

This sets the value of COLOR to 13, and any points or lines plotted will be in yellow. It is possible to change the color as often as desired within a program by reassigning a new value to COLOR.

At last, we are ready to place a brick in a particular location. To do this we use the PLOT statement. To plot a point in column 10, row 2, the statement would be:

```
30 PLOT 10,2
```

The statement PLOT X, Y would graph a single brick in column X, row Y. To summarize, our program would look like:

```
10 GR
20 COLOR = 13
30 PLOT 10,2
```

Upon execution, the screen would have a single rectangular brick in yellow plotted on location (10,2).

Now using our Color Table, write a program that will place a white dot in the middle of the screen.

```
10 GR
20 COLOR = 15
30 PLOT 19,19
```

Was this it? If not, you might have had: 30 PLOT 20,20. Your mistake was in not remembering that the twentieth column and row are numbered 19 since the numbering starts at 0. The program below was designed to test your understanding. What will the output of this program be?

```
10 GR
20 COLOR = 2
30 PLOT 39,0
40 COLOR = 12
50 PLOT 16,32
```

A blue point in the upper right hand corner and a green point in location (16,32) should have been demonstrated on the blue screen. Perhaps you can anticipate what will be involved in placing the HAPPY FACE eyes!

The second step toward our artwork will be the ability to draw lines. If you were asked to come up to the blackboard and draw a line, you would need three bits of information:

A.) Should the line be vertical or horizontal?
(The Apple cannot draw diagonal lines in LO-RES)
B.) How long should the line be?
C.) Where should the line be placed?

The Apple needs exactly the same information. The statement to draw a horizontal line might look like this:

```
30 HLIN 3, 24 AT 21
```

This HLIN statement draws a horizontal line from column 3 to column 24 at the horizontal row position 21. It plots a line from (3,21) to (24,21). The entire line will be in the assigned color. Before moving on, trace the output of this program.

```
10 GR
20 COLOR = 6
30 HLIN 0,39 AT 15
40 COLOR = 8
50 HLIN 4,15 AT 39
```

The VLIN statement works similarly to the HLIN but yields a vertical line.

```
30 VLIN 2,16 AT 0
```

A vertical line from row 2 to 16 at column 0 will be outputted.

Visions of face and smiling mouth should be dancing in your head, but a few particulars before we finish our HAPPY FACE. First, we can use variables or mathematical expressions in plotting commands or COLOR statements. This fact is demonstrated by the following program:

```
10 INPUT "COLOR:";C
20 INPUT "NUMBER FROM 20 TO 39:";N
30 GR
40 COLOR = C
50 PLOT N, N - 10
60 VLIN 0, N AT N
70 HLIN N - 15, N AT 6
```

When the user inputted 11 for C and 25 for N, the output is shown at the right. A point was plotted at (25,15); a vertical line was drawn from 0 to 25 at 25; a horizontal line from 10 to 25 at 6. When writing a program, be careful to keep the value of the coordinates in the PLOT, VLIN, and HLIN confined to integers between 0 and 39. Any other value will cause a STOP in the program with an OUT OF RANGE ERROR.

Another question you might be thinking of: "I wonder what happens to the color on the screen if one line intersects with another?" What happens to that intersection where a blue line and a red line meet? Is the result purple, or red or blue, or an error? For example, what happens to the point (15,15) where these different colored lines meet?

```
10 GR
20 COLOR = 2
30 VLIN 0, 39 AT 15
40 COLOR = 1
50 HLIN 0, 39 AT 15
```

The answer is that point (15,15) is red, the same color as the horizontal line. To answer why, think back to the discussion of storing the value for the variable COLOR. When line 40 is executed, the value 1 is placed in the mailbox, COLOR. Hence all of HLIN is drawn in color 1 -- red.

The final question relates to our upcoming work. Within a graphics program, we can use any of our programming techniques or any of the functions. One solution would be to color the entire HAPPY FACE, not just an outline. We could do this with many different line commands, each with a new horizontal location.

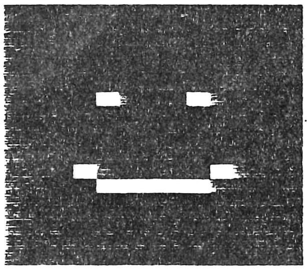
```
10 GR
20 COLOR = 12
30 HLIN B, 15 AT 3
40 HLIN 3, 15 AT 4
50 HLIN 3, 15 AT 5
ETC.
```

Instead what is called for is a loop. To draw the horizontal line we can establish a loop where the horizontal position is incremented.

```
10 GR
20 COLOR = 12
30 FOR X = 3 TO 20
40 HLIN 3, 15 AT X
50 NEXT X
```

With our solid block drawn by the program above, we need only to draw in the features. They must follow these statements so that they are not covered by the solid face. Our final program looks like:

```
10 GR
20 COLOR = 12
30 FOR X = 3 TO 20
40 HLIN 3, 15 AT X
50 NEXT X
60 COLOR = 11
70 PLOT 7,9
80 PLOT 11,9
90 HLIN 7, 11 AT 15
100 PLOT 6,14
110 PLOT 12,14
```



Our Apple HAPPY FACE just touches the surface of the possibilities of using the graphics capabilities. Smile when you say that!

EXERCISES 5.1

1. Everyone likes to see their name in lights. Write a program to output your initials in large block letters. And just to set it off, create a frame around it. Be creative with the colors.

2. The following program asks a player to roll a die by inputting "ROLL". The computer similates the throw, of course. Note the delay loops before the output and the return to INPUT. Modify this program so that when the computer rolls a die, the top face is shown on the LO-RES graph.

```
10 INPUT "TO ROLL THE DIE, TYPE ROLL";RS
20 IF RS<> "ROLL" THEN 999
30 LET X = INT(RND(1) * 6) + 1
40 FOR D = 1 TO 1000
45 NEXT D
50 PRINT X
60 FOR D = 1 TO 500
65 NEXT D
70 GO TO 10
999 END
```

3. We can combine the RND generator with graphics to produce colorful effects. Write the Applesoft statement to do each of the following. Be conscious of the range for each.

A.) Select a RND color.
B.) Graph a RND brick.
C.) Draw a RND length vertical line at 15.
D.) Draw a full horizontal line at a RND location.
E.) Select a RND color, (excluding black (color 0).

4. A SPLASH OF COLOR is our favorite program for highlighting LO-RES graphics and the RND function. Write a program to fill the graphics screen with a splash of colorful points. Plot 1000 random points in random colors (again except black).

If you get an opportunity to run this program, do so. It will give you a feel for what tandomness is. Find a point which has not been plotted (black) even after one thousand numbers.

5. This program places a large PLUS SIGN in the center of the LO-RES screen. Note the intersection (19,19) is in a different color from the rest. Modify this program so that the center of the PLUS SIGN can be inputted as an ordered pair. Then draw it at the given location. Be sure to consider what values can be inputted without causing an OUT OF RANGE ERROR.

```
5 GR
10 COLOR = 2
20 PLOT 19,19
30 COLOR = 3
40 HLIN 19 - 5 TO 19 + 5 AT 19
50 VLIN 19 - 5 TO 19 + 5 AT 19
```

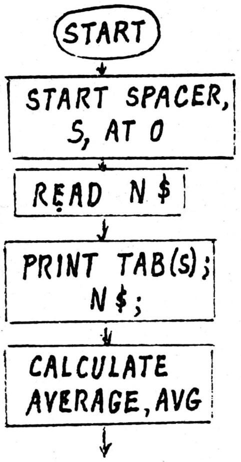
SECTION 5.2 DOING IT UP WITH LO-RES

We have shown you the tools that can be used in LO-RES GRAPHICS -- PLOT, VLIN, HLIN, COLOR. But the possible creations using these rudimentary brush strokes is limitless. Therefore we would like to demonstrate some of these creations using LO-RES. The first of these is a mathematical application -- bar graphs. The second is a pattern-making application -- a moving tunnel.

In a previous section we were able to write a program to find the average of several lists of numbers. We can now use the averaging program along with LO-RES graphics to produce a bar graph to compare these averages. The NFL is concerned with the lack of offense in today's football. Prior to considering rule changes, a study was to be conducted. The average number of points scored in the last four games by the teams in the League were to be calculated and graphed. The information would allow League officials to investigate the kind of offense being run by the high-scoring teams. Here is the team information:

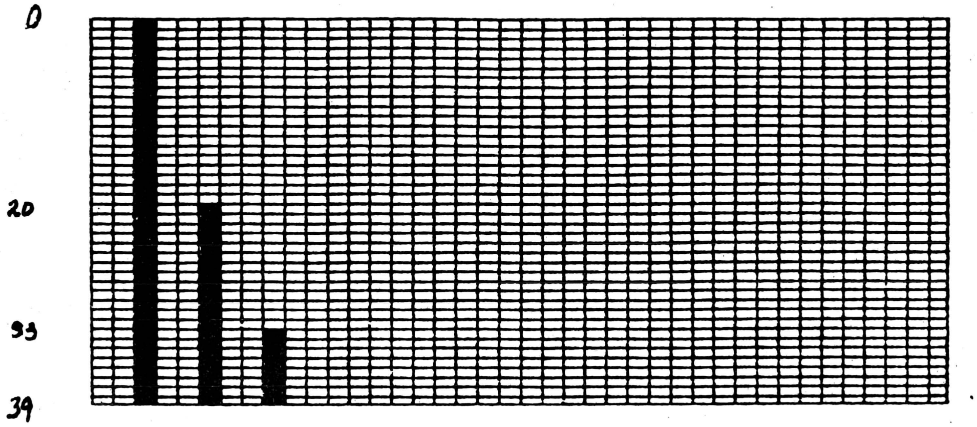
DAL - 14,28,42,27
NYG - 0,10,13,7
CHI - 3,13,9,14
BAL - 35,18,14,9
MIN - 28,35,27,24

This program is again a two part one. The first part consists of calculating the averages. Fortunately, we have already built the framework for this (see Section 3.5). But within this framework other factors will need to be considered. It is important to output the team names, even though the average will not be outputted. And of course the name should be placed under the vertical bar corresponding to the team average. We will be able to use the four text lines at the bottom of the screen to do this. These requirements indicate the need for two of our tools -- the use of TAB and the need for a counter. The counter will move the tab over each time another team name is outputted without leaving the line. A flowchart for the first part might look like this:



We have taken the liberty to include an entire looping, accumulating process in the statement "CALCULATE AVERAGE A", to conserve space.

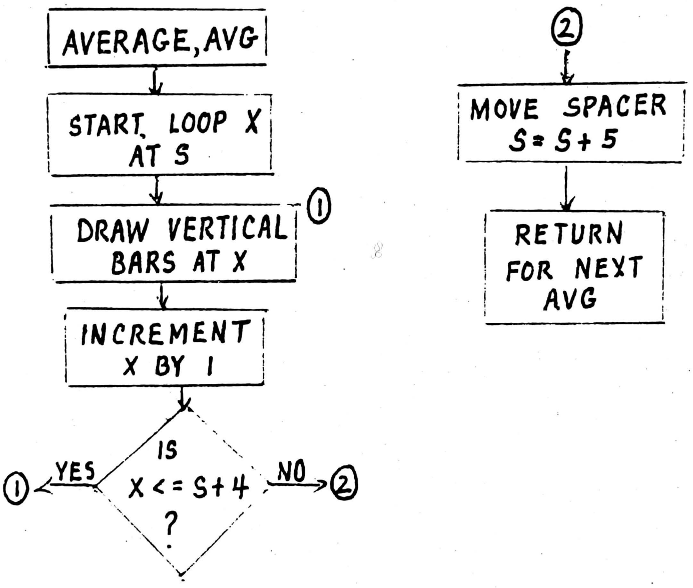
The next part is the actual bar graph. We wish to draw a vertical bar graph ending at the bottom of the screen and reaching a height according to the average. For example, let's look at the diagram below.



If an average score was 40, the bar should go from 0 to 39; for 20 the bar should go from 20 to 39; for 7, it should go from 33 to 39. What then will be the length of the bar if the average is A? The technique involves a little algebra. We need to find the portion of the 40 squares that should remain. The expression for that portion is 40-A. If A is 40, 40-A is 0; if A is 7, 40-A is 33. Because the ending point for our graph is always 39, the bottom line on the screen, the following statement generates the proper height for each bar.

```
VLIN 40-A,39
```

This statement creates a bar that is one brick wide, but we want our bar to be wider than this. A good choice here might be a width of 5. That means five vertical lines for each bar. And of course that involves a loop. But where should that loop begin and end? If you were following our first part discussion, we already established a spacer for the TAB used in outputting the teams. We can also use that as our guide for starting the bar. The loop will start at the S location and go over to the S + 4 location, making it five VLIN's wide. And the next team to be outputted, S, must be incremented by 5's to be ready for the next bar to be graphed. The first bar will go from 0 to 4; the second 5 to 9; the third 10 to 14, etc. The second half of our flowchart is below.



The program will look like the following:

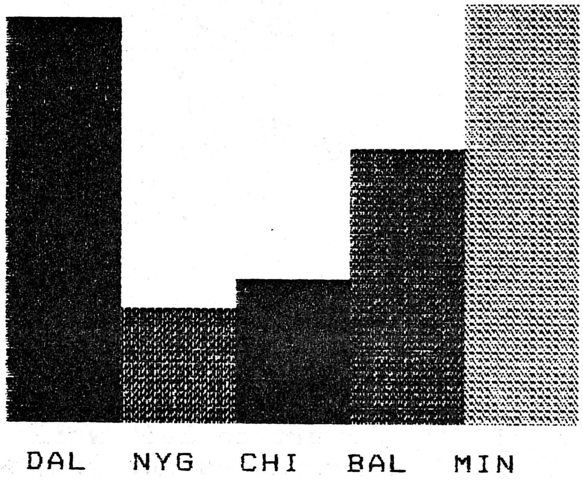
```
10 GR
15 PRINT
16 PRINT
20 LET S = 0
30 COLOR = 10
40 READ N$
41 DATA "DAL",14,28,42,27,-1
42 DATA "NYG",0,10,13,7,-1
43 DATA "CHI",3,13,9,14,-1
44 DATA "BAL",35,18,14,9,-1
45 DATA "MIN",28,35,27,-1
46 DATA "STOP"
48 IF NS = "STOP" THEN 999
50 PRINT TAB(S + 2);NS
55 LET COUNT = 0
60 LET SUM = 0
70 READ SC
80IF SC = -1 THEN 110
90 LET COUNT = COUNT + 1
95 LET SUM = SUM + SC
100 GO TO 70
110 LET AVG = SUM / COUNT
120 FOR X = S TO S + 4
130 VLIN 40 - AVG, 39 AT X
140 NEXT X
150 LET S = S + 5
160 GO TO 30
999 END
```

We moved each team name to be centered under the bar graph, by tabbing over to S + 2 each times. Of course, the GR and COLOR statements must be part of the program.

An interesting change might be to change the color for each new bar. This can be done by allowing the Apple to randomly select the color. Since the colors are numbered 1 to 15 (avoiding 0 -- black), we can write a staement to choose a random color.

```
30 COLOR = INT(RND(1) * 15) + 1
```

With this line 30 replaced, the output is shown below:



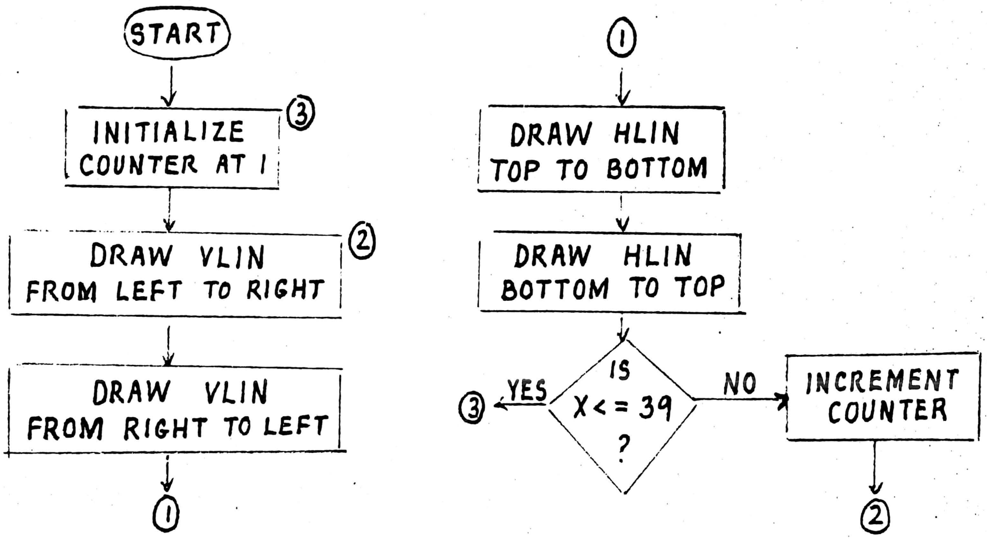
We have attempted to avoid the endless loop, but we could use this to our advantage to create optical MOVING ART. We can fill the screen with points or lines via a loop. If we then repeat that process, while changing colors, the graphics screen shows constant change occuring. That change to our eye is motion. Just to give you an idea of this technique, trace through the following program (running it would be even better).

```
5 GR
7 COLOR = INT(RND(1) * 15) + 1
10 FOR X = 0 TO 39
20 VLIN 0, 39 AT X
30 NEXT X
40 GO TO 5
```

Line 10-30 fills the screen with forty vertical lines. The entire screen is one color. Line 40 sends us back, changing the color, and repeating the screen fill. What appears is a right hand moving wave of color after color. Such a simple program has such a dramatic effect!

Our creation in the next problem will be a moving tunnel. We want to create a narrowing and then widening of a rectangular pattern. We will be running through an endless series of undulating nightmares.

The technique is a multiplication of our previous program. We can create our effect by having vertical lines moving from far left to right and far right to left. Simultaneously we wish to narrow the rectangle by having horizontal lines moving from both the top and the bottom. All of these can be done in a single loop. This loop will completely fill the scree. An unconditional transfer will create the endlessness.


START
INITIALIZE COUNTER AT 1
③
DRAW VLIN FROM LEFT TO RIGHT
②
DRAW VLIN FROM RIGHT TO LEFT
①
①
DRAW HLIN TOP TO BOTTOM
DRAW HLIN BOTTOM TO TOP
IS X <= 39 ?
YES
③
NO
INCREMENT COUNTER
②

In addition, we can change the color for each of the four lines. The only tricky part is drawing the VLIN from right to left while still using a loop which goes from 0 to 39. Line 40 demonstrates our program.

```
20 FOR X = 0 TO 39
30 VLIN 0, 39 AT X
40 VLIN 0, 39 AT ____
```

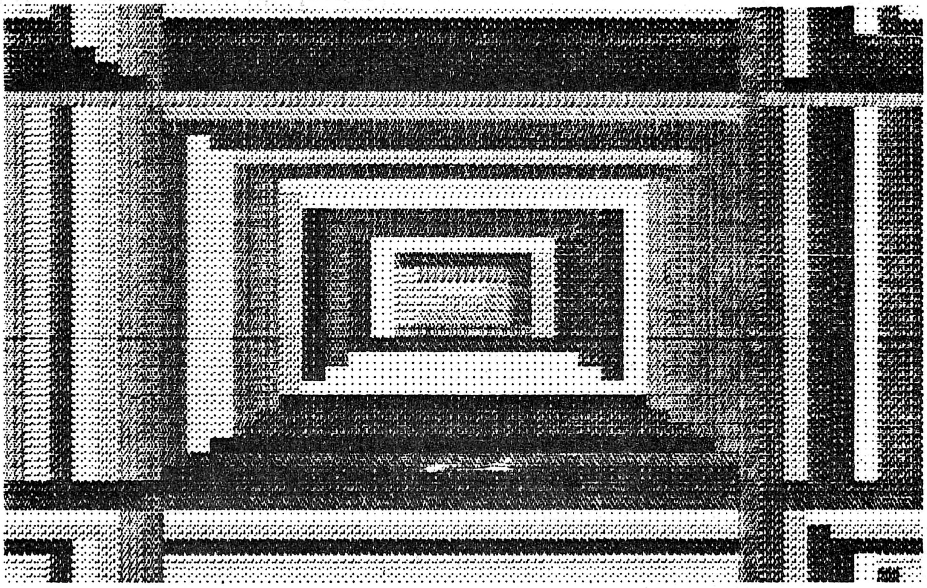
Line 40 demonstrates our problem. We want the location to go 39,38,37,36,etc. But X goes from 0 to 39. Once again we can use the algebraic expression which describes the remaining portion. When X is 0, our expression should return 39: 39-0. When X is 1, we should return 38: 39-1. Therefore our location indicator is 39-X.

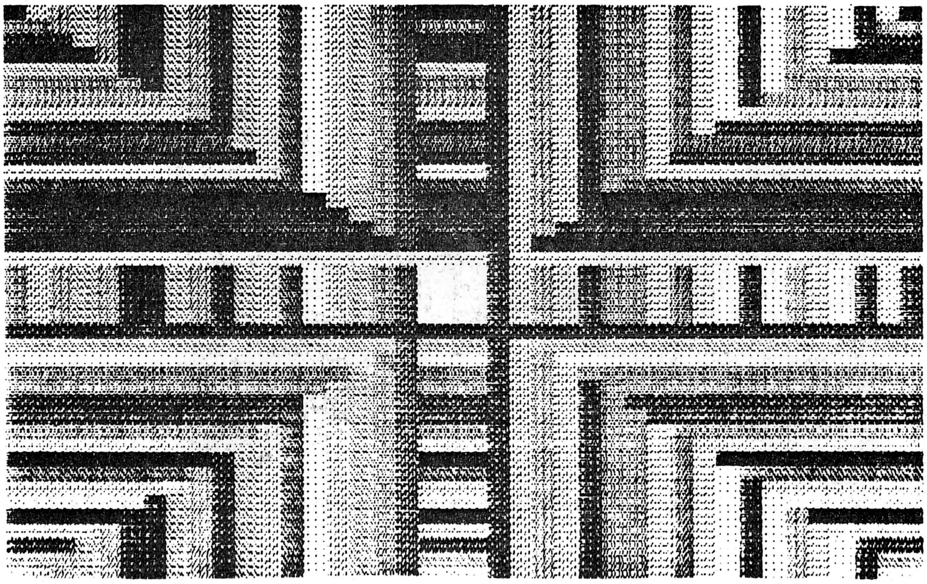
```
40 VLIN 0, 39 AT 39-X
```

Adding our horizontal lines and the random color selections, our program looks like this:

```
10 GR
20 FOR X = 0 TO 39
25 COLOR = INT(RND(1) * 15) + 1
30 VLIN 0, 39 AT 39 - X
31 COLOR = INT(RND(1) * 15) + 1
32 VLIN 0, 39 AT 39 - X
35 COLOR = INT(RND(1) * 15) + 1
40 HLIN 0, 39 AT X
41 COLOR = INT(RND(1) * 15) + 1
45 HLIN 0, 39 AT 39 - X
50 NEXT X
60 GO TO 20
```

There is no way for us to show the movement on the printouts. Below are pictures of two moments when we stopped the program with CTRL-C. Take some computer time to execute the program. It is well worth the short time to generate such visual excitement.





A little experience with the manipulation of the expressions in PLOT, VLIN and HLIN, and you will be able to construct wonderful images. The images may not be Impressionistic scenes, but they do speak of the wonder of pattern, sequence, and form which comprises the world of mathematics.

EXERCISES 5.2

1. Controlling the color on the LO-RES screen can be visually gratifying. Write a program which will fill the screen with forty lines of random color. You might modify this slightly if you wish to see the screen fill repeat.

2. The three programs below produce output on the LO-RES screen. Trace each until you are able to predict the final output. Then sketch and describe how the output occurs.

A.)

```
10 GR
20 COLOR = 3
30 FOR X = 0 TO 39
40 VLIN 0, X AT 39
50 NEXT X
```

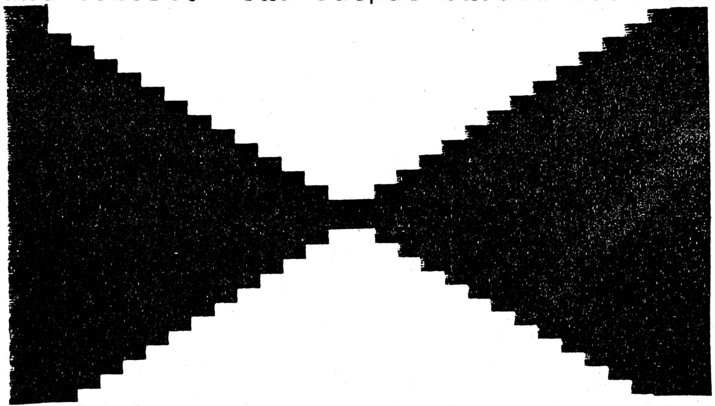
B.)

```
10 GR
20 COLOR = 4
30 FOR X = 0 TO 39
40 VLIN 0, X AT 39 - X
50 NEXT X
```

C.)

```
10 GR
20 COLOR = 2
30 FOR X = 0 TO 31
40 HLIN 0, 39 - X AT X
50 NEXT X
```

3. Using the experiences in problem #2, write a program to generate a bow tie -- in vibrant colors. The output should look like this:



Personally, we wouldn't get caught dead wearing this BOW TIE!

4. The bar graph in the Section was a VERTICAL one. Write a program to produce a HORIZONTAL bar graph. Measure the length of each classmate's arm -- from armpit to fingertip. Chart that information to compare these lengths. Each horizontal bar should be one bar wide and should begin at the far left of the screen. (If you want, make up your own problem, obtaining appropriate data!)

5. Piet Mondrian's brush covered the canvas in overlapping rectangular blocks. His art has inspired this program. You are going to make the LO-RES screen your own personal canvas. Write a program to allow the user to paint one rectangular area at a time by inputting its color and the coordinates of its four corners. The text asking for additional brush strokes should appear in the four bottom lines of the screen. And by all means, do not let this abstract creation go unnamed!

AT THE KEYBOARD SECTION 5

PADDLING DOWNSTREAM

The Apple allows us multiple ways of inputting data into our program. The information can be typed in at the keyboard through an INPUT statement or it can be built into the program by use of a READ or a LET statement. An unusual alternative to these is the use of the game paddles that accompany the Apple. These paddles can input numerical values depending on the position of the controller knob. If the knob is turned completely counter-clockwise to the off position, The output is 0. If the knob is turned to the far right, the output is 239. Thus the output is a range of integers from 0 to 239, a value controlled by the position of the knob.

We use a LET statement to assign the paddle value to a variable. Look at the program below:

```
10 LET X = PDL(0)
20 PRINT X;SPC(2);
30 GO TO 10
```

Line 10 assigns the numerical input of the first paddle, paddle zero, to the variable X. Again, this value indicates the position of the controller knob. The abbreviation PDL stands for PADDLE and the argument is the paddle number. If we wanted the value of X to depend on the position of the second paddle, line 10 would say:

```
10 LET X = PDL(1)
```

With the GO TO statement included, this program is a constant monitor on the game paddle position. When executed, it would print out, very quickly, knob position. Then it will go back and get a new paddle position (whether it had been changed or not) and print that value. Take a moment to run the program and see how the knob position affects the assigned value. Here is a partial output from our execution:

```
154 192 229 137 132 76 146 179
174 122 143 200 86 193 82 17
6 113 223 96 134 134 134 135 56
98 58 29 186 115 62 0 0 0
1 4 5 15 23 53 105 200 239 198 56
0 100 150 230 239 239 239 239
239 190 189 209 100 87 35 2 1 0
```

As we have seen, the graphics mode allows us to draw pictures and create designs by plotting points on the monitor screen. This process, although not difficult, can be very time consuming. We have also discussed that through programming skills, there is usually an easier way to perform a given task. Drawing a picture is no exception. The use of paddles can make this procedure into quite an artistic endeavor.

In using the paddles for drawing, we must break our discussion into two separate sections, one for LO-RES and one for HI-RES graphics. Beginning with LO-RES graphics, the following program would place a dot on the screen.

```
10 GR
20 COLOR = 3
30 LET X = PDL(0)
40 LET Y = PDL(1)
50 PLOT X,Y
```

Line 10 assigns X a numerical value corresponding to paddle one; line 20 assigns Y a numerical value corresponding to paddle two; line 30 plots the point X,Y. This program is error-free until either of the knobs is turned passed a setting of 39. The result in this instance would be an overflow error.

The solution to this problem is not difficult. We must limit the output range of each paddle to between 0 and 39. Therefore, before we assign either variable X or Y a value, we must divide the paddle output by 7, a factor that results in an approximation of 39 (239/7 = 34). Our revised program is:

```
10 GR
20 COLOR = 3
30 LET X = INT(PDL(0)/7)
40 LET Y = INT(PDL(1)/7)
50 PLOT X,Y
```

With this alteration, regardless of the position of the controller knob, the program will not generate an overflow error.

We could also use this technique for drawing lines on the screen. For example:

```
10 GR
20 COLOR = 3
30 LET X = INT(PDL(0)/7)
40 LET Y = INT(PDL(1)/7)
50 HLIN X,Y AT X
60 VLIN X,Y AT Y
70 HLIN 3,7 AT 6
```

Lines 30 and 40 assign X and Y a value according to paddles one and two respectively. Line 30 draws a horizontal line from the value of X to the value of Y at X's position, whatever that happens to be. Line 40 has a similar output to line 30 with the exception that a vertical line is being drawn. Finally, line 50 will draw a line on the screen from 3 to the value of Y at row six. We could add one line to the program that will enable us to perform this task continually.

```
80 GO TO 30
```

Before moving on to HI-RES graphics, there is one more point that we must discuss. What would be the result of line 30...

```
30 HLIN X,Y at X
```

...if the value of X exceeded the value of Y? For example, due to the position of controller knobs, we received an input of:

```
30 HLIN 6,2 AT 6
```

The result is not a syntax error, for the Apple is forgiving in this instance. What takes place is that, before the line is drawn, the numbers 6 and 2 are reversed such that the line reads:

```
30 HLIN 2,6 AT 6
```

Happily, the Apple draws a horizontal line from column two to column six at row six.

With one major exception, there is no difference in using the paddles for value generating in LO-RES and HI-RES graphics. In high resolution graphics, the range of numbers for the X coordinate, the horizontal coordinate, is 0 to 279. Because the paddle range is from 0 to 239, it is not possible to generate an overflow error in the X direction. The Y coordinate is another story. In this case, the range of possible values is from 0 to 159. Taking into account that the paddle generates values over the maximum Y value, we must limit our output. If we divide each Y value by 2 our range is reduced to 0 to 115. If 2 is used as a decrease factor, we are wasting quite a large number of values that Y could have, specifically, 115 to 159. Since we want a larger set of values, we can divide by a smaller rational number, like 1.5. Dividing 239 by 1.5 yields approximately 158. Then 1.5 can be our decrease factor.

```
10 HGR
20 HCOLOR = 3
30 LET X = PDL(0)
40 LET Y = INT(PDL(1)/1.5)
50 HPLOT X,Y
```

As in LO-RES graphics, the paddles can also be used to draw lines. To really demonstrate your artistic skills we can create a sketching board using the following program.

```
10 HGR
20 HCOLOR = 3
30 LET X = PDL(0)
40 LET Y = INT(PDL(1)/1.5)
50 HPLOT X,Y
60 GO TO 30
```

Notice that line 60 sends the program back to line 30. If the jump was to line 10, the screen would have been cleared each loop through the program and our purpose defeated.

Using the game paddles provides for rapid and flexible numerical input. Video game players have seen how this works. With a little practice, you can make the paddles work for you.

SECTION 5.3 MAGNIFYING THE POSSIBILITIES

One of the most sophisticated features of the Apple is its HIGH RESOLUTION graphics mode. There are many similarities between LOW and HIGH Resolution Graphics. But the primary difference is the ability to draw in more detail. While the LO-RES screen is divided into a 40 x 40 grid, the HI-RES graphics screen is divided into a 280 x 160 grid. The resulting difference is analagous to using a sharp pencil versus a thick magic marker to draw pictures.

For 15 years the National Broadcasting Corporation began its program with the peacock -- in full living colored plumage. But to indicate its changing face for the eighties, NBC searched for a new logo. It is estimated that over a million dollars went into the design of its new symbol "   ". We could have done it on the Apple for a lot less!. Write a program to create the NBC logo.

The commands in HI-RES graphics are the same as the statements in LO-RES, with the exception of the added first letter, H. The statement to switch to the HI-RES page is HGR.

```
10 HGR
```

The HGR command divides the screen into the 280 x 160 grid while leaving a small text area at the bottom portion of the screen. Likewise the statement HCOLOR is used instead of COLOR. However, there is a range of only eight colors, numbered 0 through 7. Below is a table of the colors and their corresponding numbers.

BLACK	0	BLACK	4
GREEN	1	RED	5
PURPLE	2	BLUE	6
WHITE	3	WHITE	7

The statement HCOLOR = 2 will result in points and lines drawn in purple.

To plot a single brick, the statement is HPLOT. Again this command is followed by an ordered pair of numbers corresponding to the location of the brick.

Armed with these tools, let's write a short program for placing a point in each of the four corners of the screen, each in a different color. Here's our program:

```
10 HGR
20 HCOLOR = 1
30 HPLOT 0,0
40 HCOLOR = 2
50 HPLOT 279,0
60 HCOLOR = 5
70 HPLOT 279,159
80 HPLOT = 6
90 HPLOT 0,159
```

Up to this point, we are sure you haven't had any problems. And if things weren't easy enough, the only HI-RES statement types are the ones we have described -- HGR, HCOLOR, and HPLOT. We're sure you are asking yourself about the LO-RES line drawing statements HLIN and VLIN. We can draw lines in HI-RES, but we use only the HPLOT statement. The line notion is based on the idea of a line (we are actually speaking of a line segment) as composed of a series of points. In order to draw a line in HI-RES we need to specify the two end points. The Apple does the rest for us, by connecting these two endpoints with a series of points. The end result is a line segment, drawn anywhere on the screen.

To demonstrate, let's draw a large "X" across the screen. This task is all but impossible in LO-RES, but here it is easy. We wish to draw a diagonal line from the upper left hand corner to the lower right hand corner. We want to connect 0,0 to the point 279,159. The statement is:

```
30 HPLOT 0,0 TO 279,159
```

Notice the syntax in line 30. Draw a line from one point to the other. To draw the other half of our "X", we need to connect the upper right to lower left corners.

```
40 HPLOT 279,0 to 159,0
```

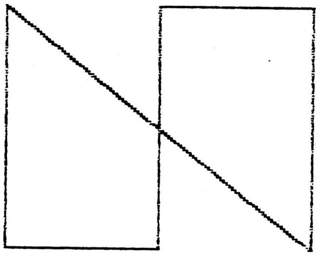
Another capacity of HPLOT can be seen in this example. Write a program to place a border around the screen. Do this before you look at ours.

```
10 HGR
20 HCOLOR
30 HPLOT 0,0 TO 279,0
40 HPLOT 279,0 TO 269,159
50 HPLOT 279,159 TO 0,159
60 HPLOT 0,159 TO 0,0
```

We can even do better than this. To draw connected lines where the starting point of one is the endpoint of another, it is possible to use the modifier <u>TO</u> as many times as needed in HPLOT. It is as if you were to draw a dot-to-dot figure, where the pencil is never lifted. Lines 30-60 can be replaced by:

```
30 HPLOT 0,0 TO 279,0 TO 279,159 TO 0,279 TO 0,0
```

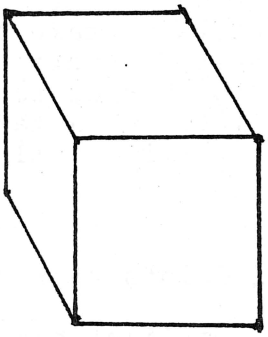
With these new tools, creating our own NBC logo will be easy. On graph paper carefully select the points and then write the short program. Our program resulted in the following output.



We can be proud as peacocks that we have gained such creative genius!

EXERCISES 5.3

1. Perspective drawing can be difficult to do, but having a piece of graph paper nearby will greatly assist you. Draw a three dimensional image of a cube and translate the information into a HI-RES graphing program. It might look like this:



2. Write a program to create a little school spirit. Draw the name or initials of your school on the HI-RES screen. Include a color loop which will redraw the same letters in all eight colors.

3. Create some waves by filling the screen in a solid color from left to right, and then refilling it in another color from right to left. Keep the bidirectional waves coming until the user pleads for mercy!

4. A piece of graph paper will help in tracing the following program. Continue to trace until you are able to predict the final output. Then describe completely how the screen will look after the program is executed.

```
10 HGR
20 HCOLOR = 2
30 LET A = 40
40 LET B = 40
50 FOR X = 0 TO 279 STEP 5
60 HPLOT X,0 TO 40,40 TO X,159
70 NEXT X
80 FOR Y = 0 TO 159 STEP 5
90 HPLOT 0,Y TO 40,40 TO 279,Y
100 NEXT Y
```

5. Make the Apple into a line plotter. Allow keyboard input of the coordinates of the starting point and the coordinates of the endpoint. Have the computer draw the line in a random color (except for black). Provide for more inputted coordinates until the user is done.

SECTION 5.4 MOVING INTO HI-RES

As we did in LO-RES, we would like to demonstrate some possible programming techniques using HI-RES graphics. With the added dimensions of oblique lines and connected line segments, the ramifications are for even more creative outputs. We are going to look at two programs. The first is a program for mathematical coordinate plane graphing using our function machine notion. The second program involves creating a shape and moving the shape across the screen.

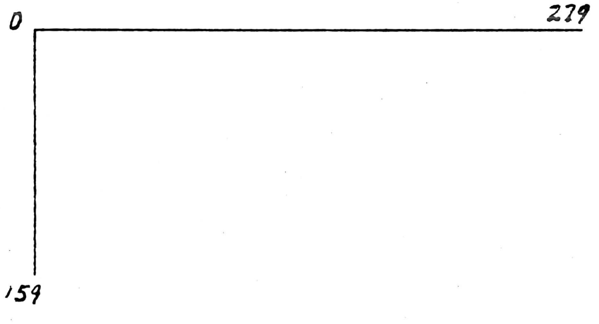
Blaise was pleased with his FUNCTION ONE robot. But we can certainly outdo him by graphing the inputs and outputs from the function machine. Write a program to graph a function in the first quadrant of a coordinate plane. Draw the x- and y-axes and plot ordered pairs in the function.

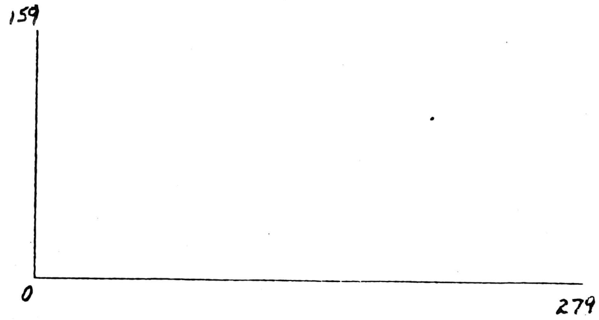
This programming problem will serve as a small example of the use of the Apple in mathematical graphing. It is in essence quite a simple program, but there are some tricks to take care of. We wish to find an entire set of inputs and outputs of the function using a loop. We can then graph each input and corresponding output as an ordered pair (X,Y). For the function rule Y = 2X + 1, we will let X take on increasing value, each time computing the Y value. Then we will simply graph the point.

```
HPLOT X,Y
```

The results will be a series of points representing ordered pairs in the function.

Unfortunately there are a few complications. First the HGR page is not set up the way we would set up the coordinate plane. Compare the grid with what the first quadrant axes should look like:

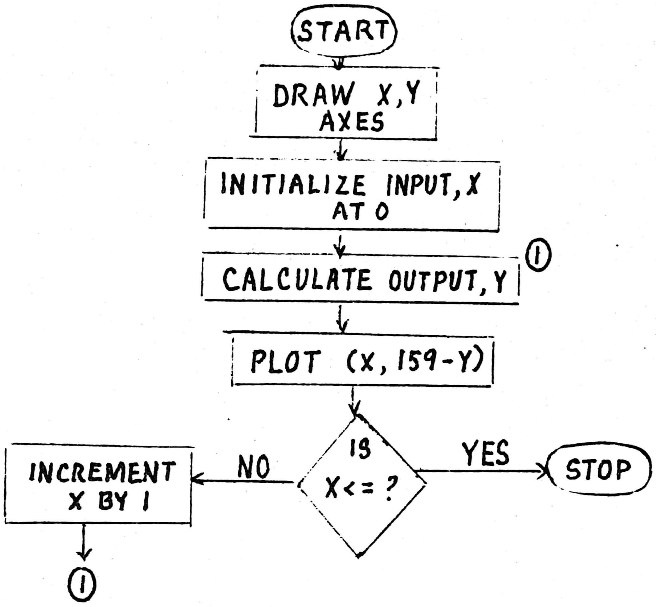




Although the x-axis (horizontal axis) goes in the correct direction, left to right, the vertical or y-axis is upside down. Therefore the graph of the function would be upside down. To correct this flaw, we will need to alter each Y value. As we have done before in previous work, the Y value will need to be reversed 159 units. The Y value to be plotted then will not be Y, but 159-Y. If Y is 0, we really wish to use 159; if Y is 16 we wish to use 143. Our plotting statement will be:

```
HPLOT X, 159-Y
```

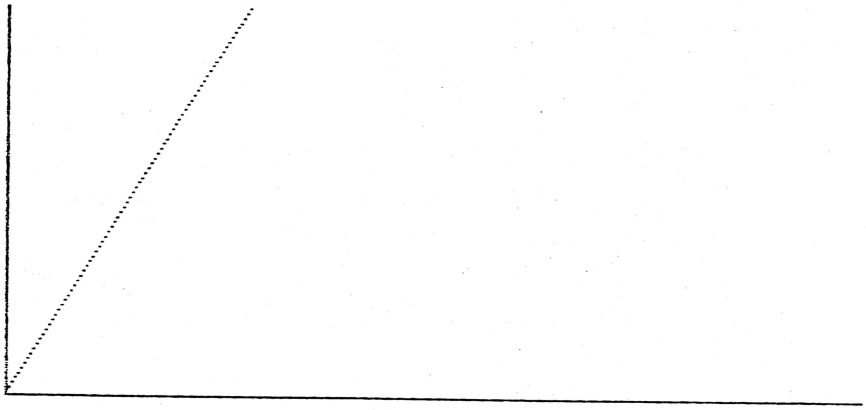
A flowchart for the problem follows:


START
DRAW X,Y AXES
INITIALIZE INPUT, X AT 0
CALCULATE OUTPUT, Y
①
PLOT (X, 159-Y)
IS X<= ?
YES
STOP
NO
INCREMENT X BY 1
①

Our flowchart indicates the other problem to be discussed. What range shall we establish for the X values? Can we loop from 0 to 279? This question will make us do come estimating before we can answer it. Any Y value above 159 will force the expression 159-Y to have a negative value. And as a result, a RANGE ERROR will end the problem. We cannot choose an X value which will produce a Y value over 159. If the function is Y = 2X + 1, then we know we must include X values less than 80. If X equals the maximum value of 79, for instance, Y = 2 * 79 + 1 or 159, a Y value that can be plotted. For the function Y = 3X + 1, X values must be less than 53. Therefore determining the test for the loop will be an important item to consider when graphing.

Here is the program for graphing the function Y = 2X + 1. Note the loop test of 79. The output is on the right hand side.

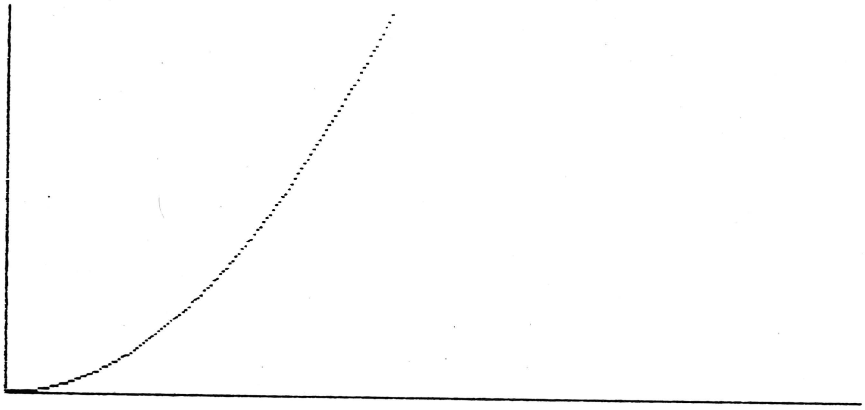
```
10 HGR
20 HCOLOR = 3
30 HPLOT 0,0 TO 0,159 TO 279,159
40 FOR X = 0 TO 79
50 LET Y = 2 * X + 1
60 HPLOT X, 159-Y
70 NEXT X
```



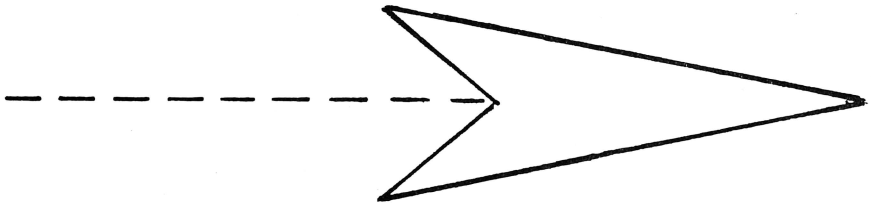
This program will serve us well to do any graphing we wish. We can change the function in line 50 and adjust the range of X in line 40. For example, if the function is Y = 1/100 * X ^ 2 what will line 40 and 50 look like? Since Y must be less than 159, Y100 * X ^ 2 must be less than 15900. Hence X ^ 2 must be less than 15900, and X must be less than SQR(15900) or approximately 125. Therefore X must be less than 15900 or 125.

```
40 FOR X = 1 TO 125
50 LET Y = 1/100 * X ^ 2
```

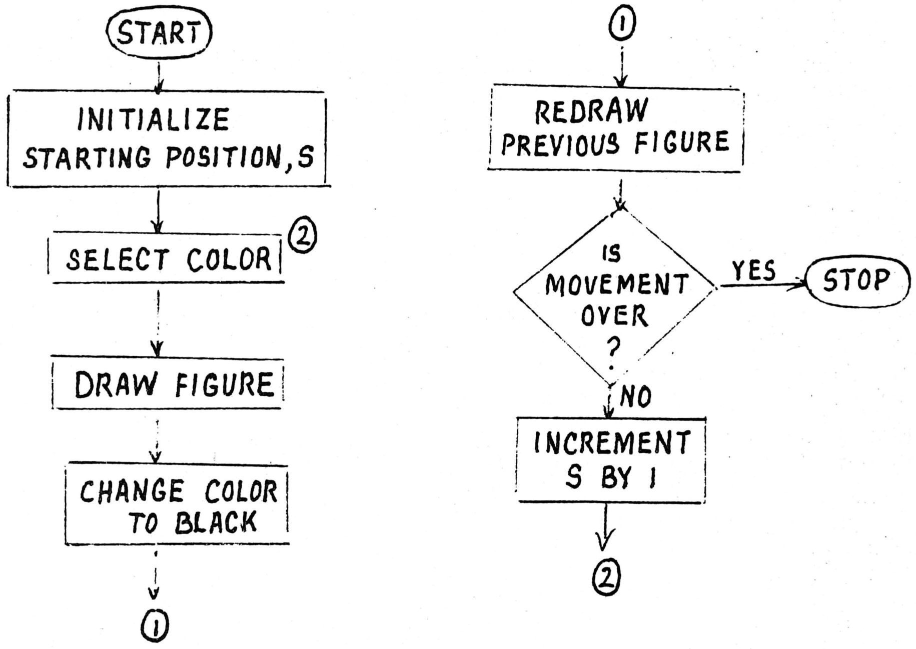
The output is a portion of a curve, called a parabola. Try your hand at inventing your own functions.



Our second problem deals with a situation which occurs in all of the well known video games. We would like to take a shape and simulate its movement across the screen. Our "FIREFOX" below should start at the left and move to the right hand along a straight path.



The notion of simulating the movement of an object on a screen involves an optical illusion. What we can do is to draw the original shape. Then we want to redraw the shape just to the right, while eliminating the first shape. This process is repeated across the screen -- redraw and blacken out. Yes, the idea of eliminating will be to copy the figure out in the color BLACK. The flowchart might look like this:


START
INITIALIZE STARTING POSITION,S
SELECT COLOR
②
DRAW FIGURE
CHANGE COLOR TO BLACK
①
①
REDRAW PREVIOUS FIGURE
IS MOVEMENT OVER ?
YES
STOP
NO
INCREMENT S BY 1
②

The first obstacle we encounter is to draw the figure so that its horizontal position may change. This involves an algebraic expression as the horizontal coordinate in HPLOT. If our shape were a single point, the statement would look like this:

```
HPLOT X,100
```

The vertical coordinate remains constant, so that the path is straight. Here is the statement to draw one FIREFOX model.

```
HPLOT 0,80 TO 40,90 TO 0,100 TO 15,90 TO 0,80
```

We want to include our loop variable as part of each first coordinate. As the loop is incremented, the first coordinate is also incremented.

```
HPLOT X,80 TO X + 40,90 TO X,100 TO X + 15,90 TO X,80
```

We are prepared to move the ship across the screen. But what about eliminating the previous one? This will take two statements. The first will change the color to black. The second will redraw the previous FIREFOX, located one unit back to the left. The ship in this position is:

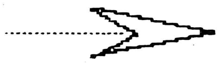
```
        HPLOT X - 1,80 TO X + 4 - 1,90 TO
        X - 1,100 TO X + 15 - 1,90 TO X,80
```

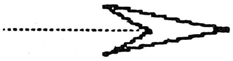
The only question left to address is the horizontal range of our flight. Is the X range from 0 to 279? Your first guess might be "yes", but we created a slight problem when we eliminate the previous ship. The HPLOT command X - 1,80 would have a negative value if X = 0. Therefore we must start our loop one unit away from the left. Also, the tip of the plane is forty units in front of the back. Since X is the coordinate of the back, we must stop forty units before reaching 279 at the back. The range therefore is:

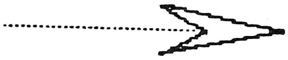
```
                FOR X = 1 TO 239
```

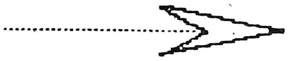
Our final program is below. And we have captured several flight moments below. We added a little tail flare to our FIREFOX. Perhaps you want to add other details. You might trace line 43 and 53 which create and eliminate the tail flare.

```
        10 HGR
        20 FOR X = 1 TO 239
        30 HCOLOR = 3
        40 HPLOT X,80 TO X + 40,90 TO X,100
           TO X + 15,90 TO X,80
        42 HCOLOR = 6
        43 HPLOT X + 5,90 TO X + 15,90
        45 HCOLOR = 0
        50 HPLOT X - 1,80 TO X + 39,90 TO X - 1,100
           TO X + 14,90 TO X - 1,80
        52 HPLOT INT((X + 5)/2),90 TO INT((X + 15)/2),90
        60 NEXT X
```









Graphic capabilities is one area where the Apple stands head and shoulders above other microprocessors. It is an area where we can combine fun with programming skills and with artistry and just TAKE OFF! It's always a real trip with LO-RES and HI-RES graphics.

EXERCISES 5.4

1. To give you some graphing experience, create your own function machine. Choose a creative function rule, but be sure to determine the correct range for the input. Then graph the ordered pairs generated on the coordinate axes.

2. Below is a program which needs serious debugging. There are syntax and programming errors. We counted eight errors. Try your hand at making this an error-free program.

```
10 HGR
20 READ N,M
30 DATA 30,50,110,110,270,100,999
40 IF N = 999 THEN 90
50 PLOT 0,0 TO N,M
60 HPLOT M,N TO 279
70 HPLOT 0,0 TO N,M AND N,M TO 279,159
```

3. Use our FIREFOX example to help with a little PONG action. Write a program to move a dot across the screen. Add another loop to move it back across the screen in the other direction.

4. To display information, mathematicians sometimes use a line-segment graph. This kind of graph is useful in recognizing and analyzing trends. Here, your task is to write a program to draw a line-segment graph to demonstrate the stock value of three companies over the past sixty years. The stock value in 1920, 1930, 1940, 1950, 1960, 1970, and 1980 for each each company should be included in a DATA statement. For IBM, the data is:

```
10 DATA 14,3,32,36,40,56,63
```

Each ten year period should represent ten units along the horizontal axis. Make 1920 the 0-year. For example, with IBM we want to plot 0,14 and connect it to 10,19, and so on. Plot each companies graph in a different color.

	1920	1930	1940	1950	1960	1970	1980
IBM	14	19	32	36	40	56	63
UNITED STEEL	12	0	3	22	30	18	11
FORD MOTOR	6	1	33	62	52	40	39

5. In our coordinate graphing work, we have shifted the axes in order to graph in the first quadrant. Rewrite the graphing program so that the graph will demonstrate all four quadrants. The origin (0,0) must be shifted to the center of the screen (139,79). Each ordered X,Y must also be shifted. Use as the function: Y = 2 * X + 1, but change the range of the first coordinates to include negative values.

A LITTLE BIT MORE SECTION 5

THE SOUND OF INPUT

At this point in our discussion, we have established three assignment commands -- LET, INPUT, and READ. There is, however, yet another command to store a value of a given variable. The <u>GET</u> statement is similar to the INPUT command in that the keyboard user determines the value of the variable. The difference between them is that after the number or phrase is keyed in, the GET command performs an automatic line feed. That is, the Apple presses return for the user and the program continues. "How will the computer know when the user is finished typing the number or phrase?", you ask yourself. It doesn't. It does, in fact, press return as soon as the first character is typed in. So only a single digit or character can be stored in a GET statement. This is certainly a restriction on our use of this statement. In addition, the digit or character is not printed on the screen as the key is struck.

Despite these restrictions, there are times when the GET command can be invaluable. One use of this command is if the user is restricted to inputting a single letter or digit. In this case, not only is the value of the variable stored, but there is not the distracting outputted answers to bog down the execution. For example:

```
60 PRINT "TYPE Y FOR YES, N FOR NO"
70 GET AS

20 PRINT "TO CONTINUE, PRESS THE C-KEY
30 GET X$
```

In this second example, when these statements are executed, line 20 prints out the information. Then the program waits for an appropriate input from the keyboard. As soon as a value is keyed in the program proceeds. The second example might continue with these lines:

```
40 IF X$ <>"C" THEN 20
50 PRINT "LET'S GET ON WITH IT."
```

Note the use of <> in line 40. This is the notation for "is not equal to". As you can see, it is possible to use the value stored in X$ by the GET statement. Here, line 40 checks for a proper response of "C" prior to continuing the program. Any response requiring more than a single character must be done using an INPUT statement.

How else might you use the GET command? We all read at different rates, so why not use the GET AS whenever lengthy instructions are given to the user. Look at the example below:

```
10 PRINT "THIS PROGRAM WILL USE ANY NUMBER"
20 PRINT "THAT YOU INPUT IN A FUNCTION MACHINE."
30 PRINT "THE OPERATIONS THAT WILL BE PERFORMED"
40 PRINT "ARE ADDITION AND MULTIPLICATION."
50 PRINT "REMEMBER ORDER OF OPERATIONS WILL BE USED."
60 PRINT "WHEN YOU ARE FINISHED READING, PRESS RETURN."
70 GET AS
80 HOME
90 INPUT X
ETC.
```

Here the user can read the text and at his leisure press any key (including RETURN) to move on. The string AS is a dummy value. Compare this procedure to the use of a delay loop. Although both create a pause, the delay loop would force the user to continue, whether finished or not.

More creative uses of the GET command are left to you!

As an aside, we would like to introduce you here to the use of sounds in programming. This is a very creative and useful little trick to make your programs a bit more interesting in special instances.

We would like to have a bell ring three times when the user gets a correct answer to the question in the program below. On the keyboard, above the G key, is the word BELL. Pressing Control G (remember, press the CTRL key and then the G key) will ring this bell. To incorporate this function into the program, we must follow the same formatting we do for any PRINT statement. Line 50 adds the BELL to our program.

```
10 PRINT "ARE YOU A PREPPIE?"
20 PRINT "TYPE A FOR ALLIGATOR IF YOU ARE,"
25 PRINT "TYPE ANY OTHER KEY IF NOT"
30 GET XS
40 IF X$ <> "A" THEN 70
50 PRINT "CTRL G CTRL G CTRL G IZOD FOREVER"
60 GO TO 999
70 PRINT "KEEP TRYING!"
999 END
```

When you use the bell in your programs, you will not be able to see it in the PRINT statement but it is there! List the program and listen for the three rings. Preppies are soooo sophisticated!

REVIEW EXERCISES

CHAPTER FIVE

1. In LO-RES graphics, we are unable to draw a diagonal line. We can simulate it by creating a series of steps in a diagonal pattern -- one down and one over. Write a program to draw a diagonal line from the top left to the bottom right of the screen.

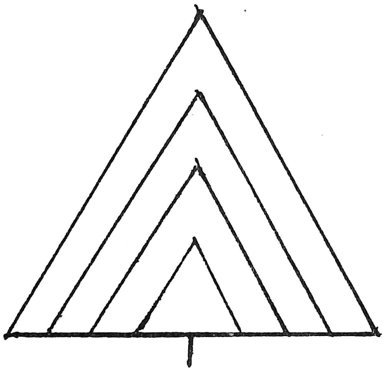
2. The following program is a design in LO-RES. Trace through the program until you are able to predict the complete outline. Describe completely what the screen will look like when execution is finished.

```
1 GR
2 COLOR = 8
10 FOR S = 0 TO 39
20 IF INT(S/2) = 2/2 THEN 70
30 FOR X = 1 TO 39 STEP 2
40 PLOT X,S
50 NEXT X
60 GO TO 100
70 FOR X = 0 TO 39 STEP 2
80 PLOT X,S
90 NEXT X
100 NEXT S
110 END
```

3. In LO-RES, create the following visual effect. Draw vertical lines starting at the far sides and coming together in the middle. Then have both sets of lines reverse direction and move back towards the sides.

4. Create a PENTAGRAM (5-pointed star) in HI-RES graphics. Draw the star with one HPLOT command. Consult some graph paper to help with the points.

5. Write a HI-RES program to show the growth of an evergreen tree. Begin with a small evergreen at the bottom of the screen. Each year, increase the size by moving the endpoints two units. Continue up to the limits of the screen.



CHAPTER 6

THERE AND BACK AGAIN

Section 6.1 An Array Of Possibilities
Section 6.2 One Vote For Subscripts
At The Keyboard: Principal Files
Section 6.3 Sorting It All Out
Section 6.4 Go-Go Practice
A Little Bit More: More If's
Review Exercises

SECTION 6.1 AN ARRAY OF POSSIBILITIES

Although we are now equipped with enough Applesoft muscle to handle almost any programming problem, we are going to explore a new technique for handling and manipulating large amounts of data. We will organize the information into lists or ARRAYS. The elements in the arrays are SUBSCRIPTED VARIABLES.

You put your money in and take your chances. You pull the arm, symbols flash by. The first window stops on "CHERRY". The second stops at "CHERRY". Crossed fingers will not prevent a "PLUM" from showing in the last window. Another lost dollar. But the "slots" are infectious and you put in another coin. You are to write a program to simulate the three windows of a slot machine. Read in the symbols and randomly select, one at a time, the three windows.

To solve this programming problem we will need to use subscripted variables. If you have had any experience with subscripts, you might remember them as small numbers beneath variables: x_1 ,x_2 ,etc. In Applesoft a subscripted value looks like: X(1), X(2) (Read X-sub2). A subscripted string variable might look like: AS(1) or AS(5). The variable part of the subscripted variable has the same restrictions as ordinary variables. The subscripts must have non-negative integer values only. Do not confuse subscripted variables with other variable names. X(2) is a subscripted variable, X2 is not.

The most important feature of subscripted variables is the way in which they are stored. The set of subscripted variables that have the same variable assignment are stored in a one-dimensional array or list.

X(0)	X(1)	X(2)	X(3)	X(4)
25	37	18	-4	5.3

It is as if the mailbox has been subdivided into a line of smaller storage boxes. The value of X(0) is 25; the value of X(3) is -4. The value of X(2) + X(4) is 23.3. The value of X(1 + 2) is -4.

The value of using subscripted variables is the power to assign a lot of values to many variables. For the subscript of a subscripted variable can be a variable. If K = 4 then X(K) = X(4). Be careful not to confuse the value of the subscript, 4, with the value of the subscripted value X(4), 5.3.

We begin by writing a program to store and output values for subscripted values

```
10 READ X(1), X(2), X(3)
20 DATA 8,-3,14                         X(1) = 8
30 PRINT "X(1) = ";X(1)                 X(2) = -3
40 PRINT "X(2) = ";X(2)                 X(3) = 14
50 PRINT "X(3) = ";X(3)
```

This program does not take advantage of the variable subscript. We can get the same storage and output placing the read and print in a loop. We will read one value and print out one value at a time.

```
10 FOR A = 1 TO 3
20 READ X(A)
30 PRINT "X(";A;") = ; X(A)
40 NEXT A
50 DATA 8,-3,14
```

When A is 1, we read 8 in X(1). The output is the string "X(", followed by A's value, 1. The string ")=" follows. Then the value stored in X(1) is printed -- 8. The loop continues by incrementing A to 2 and repeating the process for X(2).

By using a pair of consecutive loops, we could do all the storing and assigning of values in one loop and all the outputting in another.

```
10 FOR A = 1 TO 3
20 READ X(A)
30 NEXT A
40 FOR Y = 3 TO 1 STEP-1
50 PRINT "X(";Y;") = ";X(Y)
60 NEXT Y
70 DATA 8,-3,14
```

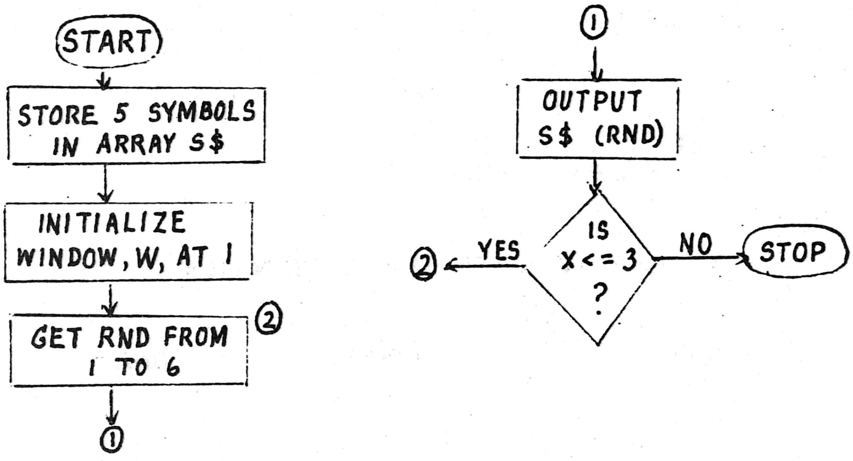
Lines 10-30 store values 8, -3, and 14 into X(1), x(2), and X(3) respectively. Once the first loop reaches its limit, the second loop begins. What is the difference in the way we output our values here? If you said in reverse order, you are on top of these new ideas! As line 40 is executed the first time, Y has the value 3. Line 50 outputs X(3) and its value 14. The next time through the loop, Y is 2 and X(2) and its stored value -3 is outputted. A tracing would look like this:

```
STORAGE VALUES                     OUTPUT
X(1)  X(2)  X(3)
 8     -3    14                    X(3) = 14
                                   X(2) = -3
                                   X(1) = 8
```

We are ready to tackle our programming problem and, we will need to use our subscripted string variables. If the strings we want stored are: CHERRY, PLUM, LEMON, BAR, BELL, try to write the portion of our program which will store these in subscripted string array SS. There are five symbols, so we would expect a loop which repeats that many times. Do this on your own before looking at ours.

```
10 FOR X = 1 TO 5
20 READ S(X)
30 NEXT X
40 DATA "CHERRY", "PLUM", "LEMON", "BAR", "BELL"
```

The rest of our program will be the actual slot machine functioning. A brief discussion before we attempt to flowchart this portion. We want to print out three strings randomly. We can do this by randomly selecting a subscript value from 1 to 5. This process simply needs to be done three times -- one for each window.


START
STORE 5 SYMBOLS IN ARRAY S$
INITIALIZE WINDOW, W, AT 1
GET RND FROM 1 TO 6
②
①
①
OUTPUT S$ (RND)
IS X<=3 ?
YES
②
NO
STOP

In translating this to Applesoft, be sure to generate the correct random number possibilities and to output the symbol names in proper format. Take time to write the program before looking at ours.

```
5 PRINT "SLOT MACHINE"
6 PRINT
7 PRINT
10 FOR X = 1 TO 6
20 READ SS(X)
30 DATA "CHERRY", "PLUM", "LEMON", "BAR", "BELL"
40 NEXT X
50 INPUT "PULL THE LEVER (TYPE COIN): "; CS
55 IF CS = "COIN" THEN 60
58 GO TO 999
60 FOR W = 1 TO 3
70 LET Y = INT(RND(1) * 5) + 1
80 PRINT SS(Y),
90 FOR D = 1 TO 2000
100 NEXT D
110 NEXT W
999 END
```

We inserted two small parts to spice up the program. Lines 50-58 were added to allow the user to start the slot machine process. What do lines 90 and 100 do? Did you recognize that as a delay loop? We wanted to put a little anxiety into the slot machine player as he anticipates the next window. Here are two executions of the program.

```
]
SLOT MACHINE


PULL THE LEVER (TYPE COIN) COIN


CHERRY          CHERRY          BELL
```

```
]RUN
SLOT MACHINE

PULL THE LEVER (TYPE COIN) COIN

BAR             LEMON           BAR
```

We have only touched on the slot machine process. Obviously we will need to check the three windows to see if they are a winning combination. And we will want to provide the winner with some reward and the loser some encouragement. These will be left as exercises. You should feel you have a handle on our one-armed array!

EXERCISES 6.1

1. If subscripts boggled your mind, we've written a program to try to help you untangle your neural pathways. Complete a two column trace of the following program.

```
10 FOR X = 0 TO 3
20 READ A(X)
30 DATA 14,20,-4,8,2,19,8
40 NEXT X
50 READ A1,A2,A3
60 PRINT A(0)
70 PRINT A(1) + A1
80 PRINT A(1 + 2)
90 PRINT A(3) + A(1)
100 PRINT A1 + A3
120 PRINT A3 + A(2)
130 PRINT A(A1)
140 PRINT A(A3/4)
```

2. Bond, 007, is back at work. This time he has intercepted a correspondence intended for an enemy agent. James thinks he may be on to a way to crack the code. He inputs the words one at a time into an array.

Then he has the computer output every other word only. The sentence is below. Duplicate this process on the Apple.

LIQUID OXYGEN ATE GLUE BOND THERE'S NO LETTER W

3. Write a program to allow ten random integers from 1 to 100, to be stored in an array. Then combine this with a program that will print out only those greater than fifty.

4. Auguste Comte, Renaissance philosopher, claimed: "There is no inquiry which is not finally reducable to a question of numbers". Certainly, sequence and pattern define a relationship between nature and mathematics. Trace this program carefully. The sequence it generates may be what Comte had in mind.

```
10 LET A(1) = 1
20 LET A(2) = 1
30 PRINT A(1); SPC(2); A(2); SPC(2);
40 FOR X = 3 TO 10
50 LET A(X) = A(X-1) + A(X-2)
60 PRINT A(X); SPC(2);
70 NEXT X
```

Investigate some of the fascinating patterns associated with this sequence -- FIBONACCI SERIES.

5. Apple isn't the only word in our vocabulary, but it is one of our favorites. Write a program to store the letters in "APPLE" in an array. Then using RND, print out a five letter combination of these. The RND will correspond to the subscript for each letter. Possible outputs might be:

```
APLLE
EAPPL
PPPPE
AAPPP
LEAPE
```

Generate twenty five such combination "words". Was it your lucky day? Was "APPLE" one of your outputs?

SECTION 6.2 ONE VOTE FOR SUBSCRIPTS

Having had some practice with subscripted variables, we are ready to tackle some problems involving arrays and subscripts. Our programming techniques and functions can be used in conjuction with subscripted variables to simplify the problem. Two problems will be explored here. One is a vote tabulation, using a subscripted COUNTER. The second is a random number grab bag, using nested loops.

With all of the new technology, election day has been shortened. With advanced polls, sampling techniques, and computerized mathematics, the public knows the results of the election before the day is over. We want to write a program to tabulate votes from our own Presidential election. The program should count the number of votes for each candidate and output totals and percentages. There are three candidates from which to choose, each assigned a number to simplify the vote tabulation.

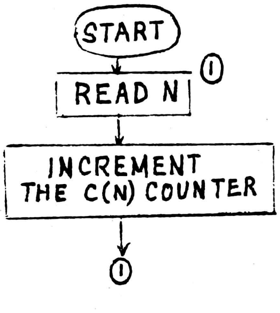
1. ALFRED E. NEUMAN
2. LASSIE
3. BIONIC WOMAN

Each polling location reports the votes as a list of numbers, where each number corresponds to the candidate's. A "2" vote, then, would be a vote for Lassie.

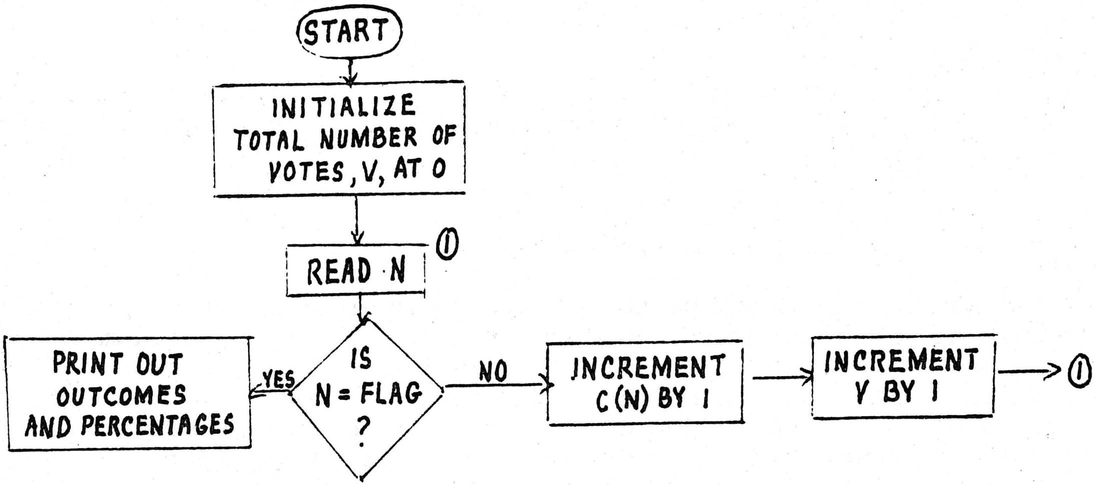
Example DATA 1,2,1,1,2,3,3,3,3,1,3,2,etc.

Tally the votes one at a time and report the outcome.

The program will certainly need a counter to do the tally work. This program before would call for the use of branching. But we will use an array. When we read in a number, we will use that as an indicator for the counter we wish to increment. The counter will be a subscripted variable, where the subscript corresponds to the candidate number. Therefore the counter for LASSIE will be C(2); for ALFRED E. NEUMAN C(1); for BIONIC WOMAN C(3). One part of the flowchart looks like:



The final portion will be to output the results. Since we want the percentage, we will need a total for the number of votes. That total can be found in two different ways. One is to simply add up the total from the three counters C(1), C(2), and C(3) at the end of the reading loop. The alternative is to insert in the reading loop a separate vote counter, incremented each time any vote is read. This second method is our choice. The final flowchart looks like:


START
INITIALIZE TOTAL NUMBER OF VOTES, V, AT 0
READ N
①
IS N = FLAG ?
YES
NO
PRINT OUT OUTCOMES AND PERCENTAGES
INCREMENT C(N) BY 1
INCREMENT V BY 1
①

The program uses 999 as a flag. We have used another subscripted loop to print the results. Take time to trace through the program to see who won the election.

```
5 PRINT "ELECTION RETURNS"
10 LET V = 0
20 READ N
30 DATA 1,1,2,3,3,3,1,2,3,1,1,1,2,3,3,3,2,3
32 DATA 1,2,2,2,2,1,2,2,1,1,2,2,2
34 DATA 3,3,3,2,3,2,2,3,3,3
36 DATA 1,3,1,1,3,2,3,999
40 IF N=999 THEN 80
50 LET C(N) = C(N) + 1
60 LET V = V + 1
70 GO TO 20
80 PRINT TAB(20);"TOTAL";TAB(30); "PERCENTAGE"
90 PRINT
100 FOR X = 1 TO 3
110 PRINT "CANDIDATE ";X;TAB(22); C(X);TAB(34);
    INT(C(X)/V * 100);"%"
120 NEXT X
140 END
```

Our second program employs nested loops to pick random numbers without repetitions.

"Who wants to be first?" This infamous teacher's line always brings a chorus of !ME!ME!ME!" Write a program to randomly pick the numbers 1 to 20, representing twenty children. Of course, we do not want any of the numbers repeated as they are chosen. The program should say:

```
HERE IS OUR ORDER CHILDREN:
10,2,19,3,8,7,5,18,17,12,11,...
```

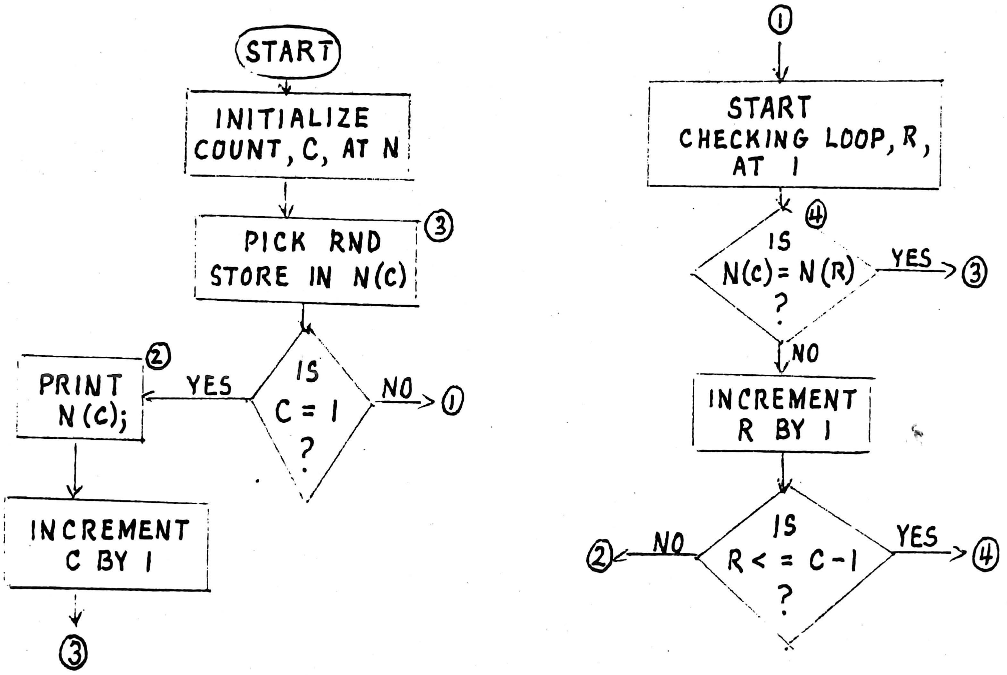
The process of picking twenty random numbers from 1 to 20 should not be a problem. It would look like this:

```
10 FOR X = 1 TO 20
20 PRINT INT(RND(1) * 200 + 1;",";
30 NEXT X
```

But we have no control over whether all the integers from 1 to 20 will be included in the list. We undoubtedly will have duplicates. Here's the output we obtained:

```
12,9,1,2,4,9,17,18,10,6,6
15,5,3,13,11,2,8,20,19
```

We need to rewrite our program to avoid a number reoccurring. The process we will follow is to generate a number and then check that number to see if it has already occurred. If not we move on. If it does we will go back and pick another random number instead. We will use subscripted variables to store each of the random numbers. We will begin the process by picking the first one. Since this cannot be a repetition we do not need to check it. After the first one, we will have to check each number. As each new number is chosen, we will compare it to each of the other ones already selected. That is, when we get the sixth random number we will see if it is the same as any of the first five. This will be the inner loop: a loop to check the new number with all the ones stored before it. The flowchart looks like this:


START
INITIALIZE COUNT, C, AT N
PICK RND STORE IN N(C)
③
IS C = 1 ?
YES
NO
①
②
PRINT N(C);
INCREMENT C BY 1
③
①
START CHECKING LOOP, R, AT 1
④
IS N(C) = N(R) ?
YES
③
NO
INCREMENT R BY 1
IS R < = C-1 ?
NO
②
YES
④

Note that the checking loop, R, needs to go from 1 up to the previous value C. Hence the test is: IF R = C-1 THEN _____. The program looks like this:

```
10 FOR C = 1 TO 20
20 LET N(C) = INT(RND(1) * 20) + 1
30 IF C = 1 THEN 70
40 FOR R = 1 TO C - 1
50 IF N(C) = N(R) THEN 20
60 NEXT R
70 PRINT N(C);
80 NEXT C
```

We ran this program and here is the output:

```
14,2,3,15,17,10,4,12,8,11
BAD SUBSCRIPT ERROR
```

A bad subscript is one that is not expected by the computer. Applesoft allows a programmer to use automatically subscripte from 0 to 10. But is we wish to use larger subscripts, like in our program, we must reserve extra space for the array. To do this, we begin our program with a DIMENSION statement.

```
2 DIM N(20)
```

The DIM statement tells the Apple how many storage locations to reserve for the subscripted variable N. Here space is made for N(0) to N(20). This process simply prevents the Apple from wasting space. It is always good to dimension the variable larger than you will need. Many programmers dimension all arrays. Therefore we can add this to our program:

```
2 DIM N(25)
```

We would like to trace this program. Sometimes with RND it is more difficult to trace. Here we have simply picked the following list as our random numbers.

12,9,1,2,4,9,17,18,10,6,etc...

The first time, when C is 1, N(1) = 12. Since this is the first value selected, 12 is outputted immediately. Then the C loop increments C to 2 and N(2) = 9. Failing the test: Is C = 1, this number enters the checking loop. Here R will go from 1 to 1, since there is only one to check against. The two numbers are not the same, so 9 is outputted.

STORAGE VALUES				OUTPUT
C	N(C)	R	N(R)	
1	12			12, 9
2	9	1	12	

Three more times through the loop and our trace appears like the one below. Note the progression of R's and N(R)'s.

STORAGE VALUES				OUTPUT
C	N8C9	R	N(R)	
1	12			12, 9, 1, 2, 4
2	9	1	12	
3	1	1	12	
		2	9	
4	2	1	12	
		2	9	
		3	1	
5	4	1	12	
		2	9	
		3	1	
		4	2	

The next time through the counter, and problems will arise. N(6) = 9.

When it enters the checking loop, R is 1 and it asks the question is N(6) = N(1)? With a NO answer, R is incremented. Now the question is, does N(6) = N(2)? Both have a value of 9. We want this N(6) value to be rejected. The conditional transfer, line 50, returns to get a new value for N(6), to replace 9. The next random number is 17. This new N(6) is submitted to the checking loop again. Since N(6) does not equal any of the values N(1) through N(5), the N(6) is accepted and outputted. This process continues until the twenty values have all been picked.

Subscripted variables can indeed allow us to perform complicated processes with ease. The number selector would be a 180 questions branching program, if it were not for arrays. Although the notation may be very new to you, some practice will place you first on the list along with the BIONIC WOMAN.

EXERCISES 6.2

1. Using an array of counters, make it easy to keep track of results. Write a program to simulate throwing a die five hundred times. The program should count how many times a certain number appears on the die. Output the frequencies in column form.

2. The following program contains a small "bug". Trace the program to help locate this bug of omission. Be sure to correct the program.

```
10 FOR X = 1 TO 15
20 LET N(X) = 5 * X
30 NEXT X
50 PRINT "NON-ZERO MULTIPLES OF FIVE"
60 FOR A = 1 TO 10
70 PRINT "MULTIPLE #";A,N(A)
80 NEXT A
```

3. Captain Apple is ready to impress your friends again. Write a program to allow your friends to input twenty five rational numbers. Some of the input should be integer values. As they complete their part, have Captain Apple output all twenty values separated into two sets -- INTEGER, NON-INTEGERS. The output should look like:

```
    INTEGERS:     2  4  9  10123
NON-INTEGERS:  1.314  -219.3  .00083
```

4. Write a program to read in ten pieces of numerical data. Then using only one loop, print out the ten values in two columns. One column will be in the order they were stored. The other should be in the reverse order.

```
5          40
1          3
12         12
3          1
40         5
```

5. In this age of liberation, the infamous "little black book" is no longer a strictly male phenomenon. Storing information and accessing that data quickly is a task that the computer handles well. Write a program which will read all the essential information on an eligible male. Four subscripted arrays should contain the boy's name NS(X), his telephone number P$(X), his eye color E$(X), and his favorite hobby HS(X). For example, the data for one individual might look like this:

```
DATA HERMAN HOLLERITH, 200-1111, BLUE, SOCIALIZING
```

Once all the data is read in, the program should allow it to be accessed. The keyboard user inputs a name and the program will print out all the information on that individual. Be sure to check for an inputted name which does not correspond to any in the black book.

AT THE KEYBOARD SECTION 6

PRINCIPAL FILES

It is a cool fall morning and the second bell just finished ringing. Suddenly, the principal of the school rushes into your classroom with an urgent problem. It seems that one of his disks was erased (he said that his secretary accidently pushed the wrong button, but actually he sat on it!) and three of his most important programs were destroyed. He had a mailing list program which contained every student's name, a grade sheet program, and a billing program. The dilemma he faces is re-typing every student's name -- three times. Needless to say, he hopes that you have a solution to his problem.

Once again, the Apple, with you as the programmer, comes to the rescue. You explain to the principal that there is a type of program that is called a text file. Unlike most Applesoft programs, he cannot run this program, nor can he perform any of the familiar DOS commands with it. The program's only use is for storing data, whether the data is composed of numbers, words, correspondences, or even names of students. You go on to explain that in order to use the names for all three of his programs, they must only be typed in one time.

Following is the listing of a program which generates a text file. Preceding the list is an explanation of each line. Try to follow along with each description so that you will be able to help our absent minded principal!

```
10 DIM A$(100)
15 LET COUNT = 0
20 LET D$ = CHR$(4)
40 PRINT "YOU GET TO TYPE ONE STRING AT A TIME."
50 PRINT "A STRING MAY HAVE UP TO 239 CHARACTERS."
60 PRINT "THIS PROGRAM LET'S YOU WRITE TEXT FILES."
70 PRINT
80 PRINT "(PRESS THE RETURN KEY TO QUIT.)
85 LET COUNT = COUNT + 1
90 PRINT "TYPE STRING #";COUNT;": ";
95 INPUT A$(COUNT)
110 IF AS(COUNT) <> " " GOTO 60
130 INPUT "WHAT FILE NAME DO YOU WANT? ";N$
160 PRINT D$;"OPEN";N$
170 PRINT D$;"WRITE";N$
180 PRINT COUNT - 1
190 FOR J = 1 TO COUNT - 1
200 PRINT A$(J)
210 NEXT J
220 PRINT D$;"CLOSE";N$
```

Line 10 sets aside enough memory space for one hundred student names by dimensioning the variable AS to 100. In line 15 the variable COUNT, which will keep track of the number of inputted names, is initialized at 0.

Let the fun begin! The command in line 20 is the initiation of a very special concept. Accompanied by the PRINT command, CHR$(4) is a flag which indicates to the computer that the next command will be directed towards the disk drive.

```
10 PRINT CHR$(4) "CATALOG"
```

If the above statement is the first line of a program, the first operation that will be carried out is to catalog the disk.

```
10 PRINT CHRS(4) "RUN HELLO"
```

If the first line of the program was this, however, the program titled "HELLO" would be run as the first step. Notice this is a way of running a program within a program. Returning to line 20, the variable DS is assigned the value of CHRS(4) so that our typing becomes less cumbersome later in the program.

Lines 40 - 80 provide to the user, instructions on how to run the program. Notice line 50 specifies that an input may be no longer than two hundred thirty nine characters in length, a function of the memory capacity of the Apple. Line 85 increments the counter by one in preparation for the first name to be inputted. Continuing, line 90 looks mean, but, as usual, it's only a wolf in sheep's clothing. The output of this line will be:

```
TYPE STRING #1:
```

The user complies in line 95 by assigning the first name to AS(1). Moving along, line 110 is a conditional test. The instruction in line 80, "PRESS THE RETURN KEY TO QUIT", is the flag being tested. If a name had been inputted, AS(COUNT) would not be equal to a space and at this point, the program would return to line 60. If on the other hand, a space had been generated by simply pressing the RETURN key, we would move on to the next line, line 130.

Let's assume that a name had been inputted so that we can loop back through the program one more time. Notice how the jump is back to line 60. At first glance, repeating "THIS PROGRAM LETS YOU WRITE TEXT FILES" and "PRESS RETURN TO QUIT" is redundant -- the user has already seen these lines. Repeating them, however, serves the important function of reminding the user how to terminate the program. Imagine typing ninety three names and forgetting how to end the program! Line 85 increments the counter by one, now at a value of 2, and we are to enter our second name under CS(2). Again, line 110 tests to see if we inputted a name or simply pressed the RETURN key. Assuming we typed in a name, back to line 60. The counter is incremented to three and the program waits for an input. This time, however, we simply press the RETURN key and on to line 130 we go.

Line 130 asks us to input a file name. Just as we had to save an Applesoft program under a specific name, text files must also have titles so that we can refer to them from other programs. Now comes the interesting part. Line 160 used the DS character assigned in line 20, CHRS(4). Remember that PRINT D$ is a disk command. Look carefully at the syntax in line 160. The command is in quotes and the variable name for the text file is out of quotes. The <u>OPEN</u> command simply means to create a space on the disk to store data. When you open a checking account at the bank, you are doing essentially the same thing -- you are reserving space in the bank to store money. Because the name of our text file was unknown before we started this program, we had to assign to it a variable address. This variable must be out of quotes when opening a text file, otherwise the file will be open under the name NS. If, for instance, we did not want to use the input option, line 130 should be eliminated and line 160 changed to the statement below.

```
160 PRINT D$; "OPEN STUDENT NAMES"
```

Moving on to line 170, we are ready to deposit names in the file, just as we deposit money into a checking account after opening it. The first deposit into the file is the number of names that will be in that file.

The reason for this step is not yet evident but its presence is imperative. All data will be deposited through a PRINT statement. Line 180 deposits the number COUNT-1. Why not simply the value of COUNT, you ask? Review line 85 -- the counter is incremented before the input of a name. Using our example, we inputted only two names and the value of COUNT was three. Therefore, line 180 subtracts one from COUNT to generate the proper number of names. Finally, line 190 initiates a FOR-NEXT loop so that each name may be placed in the file. Notice, once again, the loop goes from 1, the first name, to COUNT-1, the last name. A$(1) is the first name, AS(2) is the second name, and A$(COUNT-1) is the last. Line 200 deposits the names through the use of a PRINT statement. Finally, line 220 closes the text file that has been created. If this line is not executed, other data may inadvertantly be stored in the file. Remember, close the door behind you!

Confidently, you walk into the principal's office (who is feverishly working on removing the crease from his bent disk) and nonchalantly mention that you have performed the arduous task of generating a list of names for him on disk. Knocking his container of pencils on the floor, he leaps up in total dismay, and asks you to show him how to retrieve this list so that it may be used in his programs. Slowly, you sit down and explain the following program.

```
10 LET DS = CHR$(4)
12 PRINT "THIS PROGRAM RETRIEVES TEXT FILES."
14 PRINT "CREATED BY THE 'CREATE FILE' PROGRAM."
20 INPUT "WHAT IS THE NAME OF THE TEXT FILE?";Z$
30 PRINT D$;"OPEN";Z$
40 PRINT D$;"READ";Z$
50 INPUT LAST
55 DIM A$(LAST)
60 FOR J = 1 TO LAST
70 INPUT A$(J)
80 NEXT J
90 PRINT D$;"CLOSE";Z$
```

You start by explaining that CHRS(4) is a flag to the computer which indicates that the next command is a disk command. Assigning a value of DS to CHR$(4) as indicated in line 10 allows you to simply type DS each time you want to use the CHRS(4) function. Lines 12 and 14 give the user instructions on how to run the program. You go on to explain that line 20 assigns to Z$ the name of the text file that is to be accessed. In our case we would input STUDENT NAMES as the title. Line 30 opens the file so that we can enter information into it or retrieve information from it. In the first program, we used the WRITE command because our goal was to deposit names into the file. In this program we wish to retrieve names from the file, hence the READ function. Notice that the variable Z$ is not in quotes. If we wanted to restrict this program to read only the file STUDENT NAMES we would eliminate the input statement in line 20 and alter lines 30 and 40 to be:

```
30 PRINT DS; "OPEN STUDENT NAMES"
40 PRINT DS; "READ STUDENT NAMES"
```

Moving on, line 50 inputs a number into memory location LAST. Up to this point, the principal has been following along, but the reason for depositing a number into memory location LAST knocks him for a loop! "Where did the number come from anyhow?", he asks. You tell him to relax, the explanation is simple. In the "Create File" example you placed the student names in a text file, true. In addition, however, you placed as the first input the total number of names contained in the file, COUNT-1, to refer to at a later time. Now is the time. This number is used in line

55 to dimension AS to hold the proper number of names. We could dimension A$ to 200, assuming that this will provide plenty of memory space for the text file, but this would be very wasteful of memory. Line 60 also uses the value of LAST as a test for a FOR-NEXT loop. Line 70 is a little tricky. Before we can use the names in a program, we must input each one into a memory location. Notice we use an array to label each "mailbox". The first name is placed into location AS(1), the second into AS(2), the third into A$(3), etc. This process continues until we have covered the entire list. Finally, line 90 closes the file, a task that must be performed whenever a file is open.

You sit back in your chair, cross your arms, and smile. The principal looks confused, however. He understands that each name is in a memory location in the computer, but this doesn't help him. He needs each name printed out. You explain that the difficult process of placing each name in a memory location has been completed. To access each location, all that has to be done is to add the following lines to the program.

```
110 FOR J = 1 TO LAST
120 PRINT A$(J)
130 NEXT J
```

Line 110 starts a FOR-NEXT loop using LAST as the test value. Line 120, instead of inputting each name into a "mailbox", prints the word on the screen. We could not have used the PRINT command in line 70 to replace INPUT because when we are working with files, remember, the PRINT command deposits information. Unfortunately, we must always follow a two step process when accessing information from a text file. Input the names into a specific memory location and then print them. You continue by explaining that this list can be used in any program. To access the names, simply refer to the array titled AS.

The principal, elated, appoints you for Honors, and better yet, tells you to take the day off!

SECTION 6.3 SORTING IT ALL OUT

Before leaving our discussion of subscripted variables, we would like to sort things out. No, we don't mean clear up misunderstandings. We mean to sort out data.

Write a program to take a list of words and to place them in alphabetical order. If the list of words were: BALLOON, KITE, FRISBEE, and AIRPLANE, the program would output them:

AIRPLANE, BALLOON, FRISBEE, KITE

This process of sorting is a complex one. There are many possible methods for doing a sort. Sorting involves two processes primarily -- comparing items and exchanging pairs of items. But these processes must be repeated many times and can use much computer time. Different procedures for sorting are designed to reduce this time.

Before starting our flowchart, we need to consider the process of swapping or exchanging. If A$ = "PLEASE" and B$ = "HELP", how can these words be switched? The following program is one guess:

```
10 LET AS = "PLEASE"
20 LET B$ = "HELP"
30 LET A$ = B$
40 LET B$ = AS
```

But a careful trace will reveal a problem. In line 30, AS is reassigned a new value -- that of B$. Therefore at line 30, "PLEASE" is replaced by "HELP". At line 40, B$ receives a new value. But that new value is the value of A$ which is now "HELP". In effect, both string variables contain the same string. To answer the dilemma, we need a "value holder" for AS while we replace its value. A new holding tank is built: 25 LET CS = AS, whose only purpose is to keep the AS value in the program until it is placed in B$. Therefore these three statements become an EXCHANGE ROUTINE:

```
20 LET C$ = AS
30 LET A$ = B$
35 LET B$ = C$
```

The end result is "HELP" in AS and "PLEASE" in B$. This exchange routine will be used repeatedly in our alphabetizer.

The sorting procedure we will be using is called a <u>BUBBLE SORT</u>. A more appropriate name might be a SINK SORT. We will pass through the names several times. On each pass we will make the word occurring latest in the alphabet sink to the bottom of the list. The method will compare the first two words and exchange if necessary so that they are alphabetized. It will then look at the second and third, third and fourth. Each time swapping will occur if necessary. After that last pair the latest occurring word will be at the bottom.

```
┌BALLOON   BALLOON   BALLOON   BALLOON
└KITE     ┌KITE      FRISBEE   FRISBEE
 FRISBEE  └FRISBEE  ┌KITE      AIRPLANE
 AIRPLANE  AIRPLANE └AIRPLANE  KITE
```

The following three comparisons were made and two exchanges occurred. The word "KITE" is now at the bottom of the list. It has SUNK to the bottom. The rest of the words are not in alphabetical order. Another entire pass must be made, but this time we do not have to compare to the bottom word "KITE". The second pass would look like this:

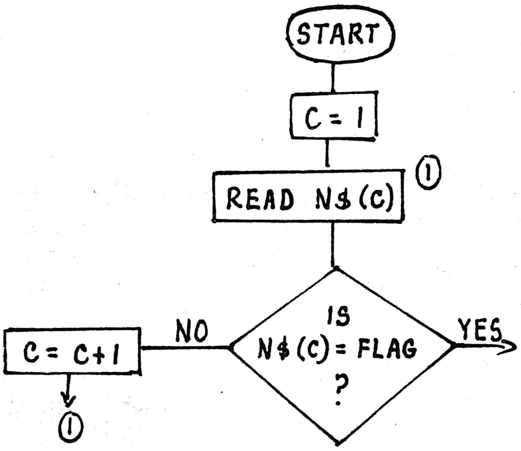
```
┌BALLOON    BALLOON    BALLOON
└FRISBEE   ┌FRISBEE    AIRPLANE
 AIRPLANE  └AIRPLANE  ┌FRISBEE
 KITE       KITE      └KITE
```

Only two comparisons were needed and one exchange was made. "FRISBEE" has fallen to its proper place at the bottom. One more pass, only one comparison, needs to be done.

```
┌BALLOON    AIRPLANE
└AIRPLANE  ┌BALLOON
 FRISBEE   └FRISBEE
 KITE       KITE
```

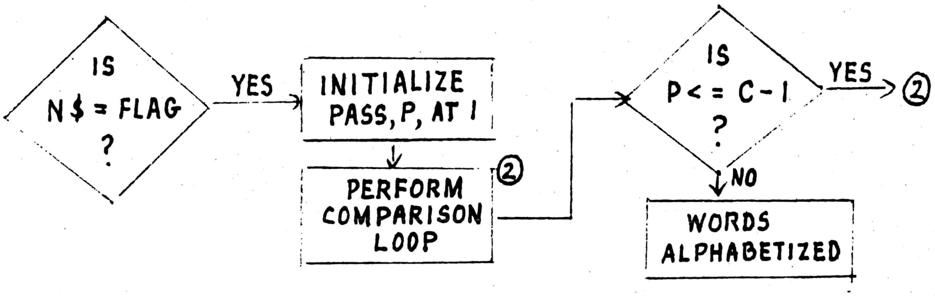
"BALLOON" has sunk below "AIRPLANE" and the words are alphabetized.

Our flowchart is comprised of three separate parts. The first consists of storing the words in an array. It looks like this:

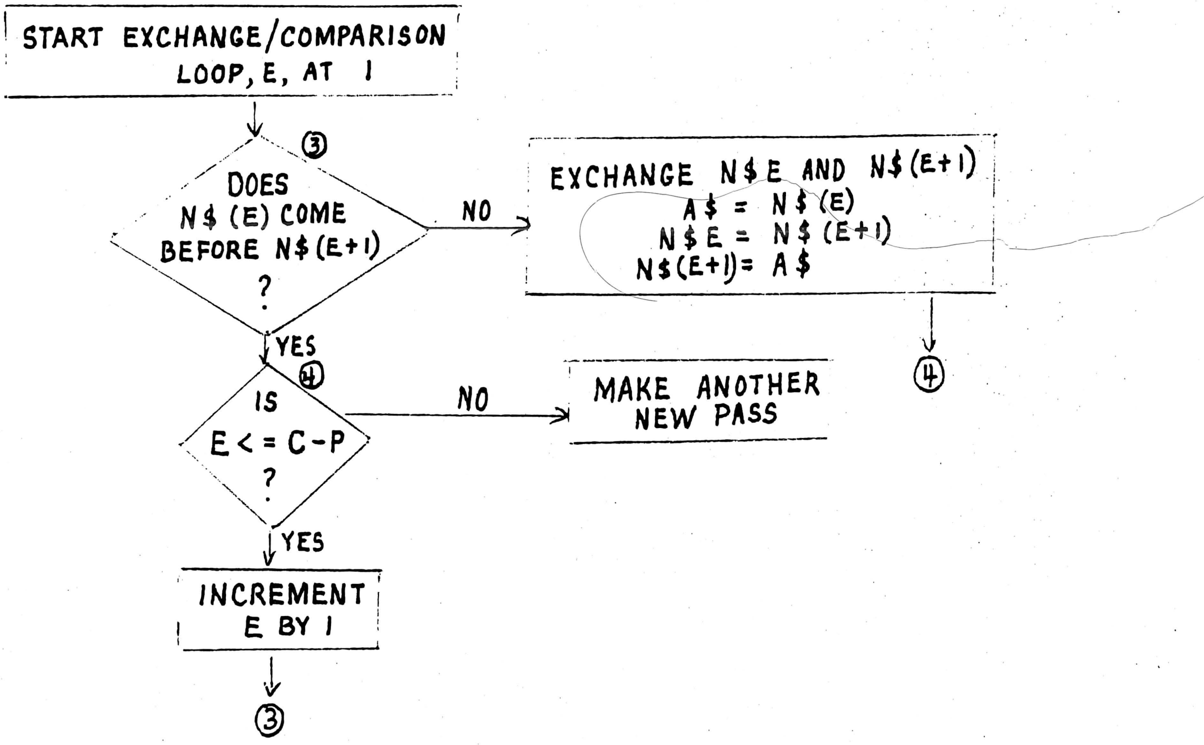


The YES path means the words are now in the array: N$(1) = "BALLOON", N$(2) = "KITE", etc. We are now ready to make the comparison/exchange passes. The actual sorting is a nested loop construction. One loop will count the number of full passes necessary. The inner loop will perform the necessary comparisons and exchanges.

The pass loop will need to occur one less than the number of words in the list. It takes one pass to get each word to the bottom, with the obvious exception of the top word. This will be in the correct order when all the others have sunk. In our program, three passes are necessary.


IS
N$ = FLAG
?
YES
INITIALIZE
PASS, P, AT 1
PERFORM
COMPARISON
LOOP
②
IS
P< = C-1
?
YES
②
NO
WORDS
ALPHABETIZED

The comparison loop will compare N$(1) with NS(2), N$(2) with NS(3) and so on. As you saw in our discussion, the number of comparisons decrease as each pass is made. In our example, we needed three, two and then one comparison. The test for the number of comparisons decreases as the passes increase. The number of comparisons is the number of words minus the number of passes. The test is C-P for each pass -- 4-1 or 3 the first time, 4-2 or 2 the second pass, so on. The flowchart for the comparison loop looks like:


START EXCHANGE/COMPARISON
LOOP, E, AT 1
③
DOES
N$ (E) COME
BEFORE N$ (E+1)
?
NO
EXCHANGE N$E AND N$(E+1)
A$ = N$ (E)
N$E = N$ (E+1)
N$(E+1)= A$
④
YES
④
IS
E < = C-P
?
NO
MAKE ANOTHER
NEW PASS
YES
INCREMENT
E BY 1
③

You might be asking how to go about asking the question: "Does NS(E) come before NS(E + 1) in the alphabet?" To help us do this, the Apple has the ability to compare two strings alphabetically using >,<. The inequality symbol, >, with strings means "precedes in the alphabet". Therefore we can simply check: IF NS(E) > NS(E + 1).

The final portion of the program prints out the new list of alphabetized words, now in N$ array.

```
2 PRINT "WORD LIST"
5 LET C = 1
10 READ N$(C)
20 DATA "BALLOON", "KITE", "FRISBEE", "AIRPLANE", "STOP"
30 IF N$ = "STOP" THEN 60
35 PRINT N$(C)
40 LET C = C + 1
50 GO TO 10
60 FOR P = 1 TO N - 1
70 FOR E = 1 TO C - P
80 IF N$(E) > N$(E + 1) THEN 120
90 AS = N$(E)
100 NS(E) = NS(E + 1)
110 NS(E + 1) = AS
120 NEXT E
130 NEXT P
140 PRINT "APLHABETIZED LIST"
150 FOR X = 1 TO C
160 PRINT NS(C)
170 NEXT X
180 END
```

```
WORD LIST

BALLOON
KITE
FRISBEE
AIRPLANE


ALPHABETIZED LIST

AIRPLANE
BALLOON
FRISBEE
KITE
```

You are not ready to retire Funk and Wagnall, but with some tracing you will feel more comfortable in dealing with this sorting notion. As a programmer of the future, the sorting procedure is an invaluable tool. And producing different strategies for sorting will be a goal.

EXERCISES 6.3

1. With our new exchange routines, the programmer can control to some degree the relationships between inputted values. For example, we want to write a program where two values A and B are inputted. But in our work, the value of A must be smaller than that of B. Write the program's beginning lines, in which you arrange a guarantee that A is smaller than B prior to starting the task at hand.

2. One approach to finding the Greatest Common Factor of two numbers is the Euclidian Algorithm. This method involves a repeated procedure of dividing divisor by remainder. The flowchart in problem #1 of Section 2.1 illustrates this Algorithm. Translate this into Applesoft. Note the exchange routine.

3. Use our newly developed sorting routine to rearrange a list of ten numbers in descending order.

4. Modify the Bubblesort to alphabetize the letters in a word. Allow the word to be entered. The output should say:

```
ALPHABET ALPHABETIZED IS AABEHLPT
```

SECTION 6.4 GO-GO PRACTICE

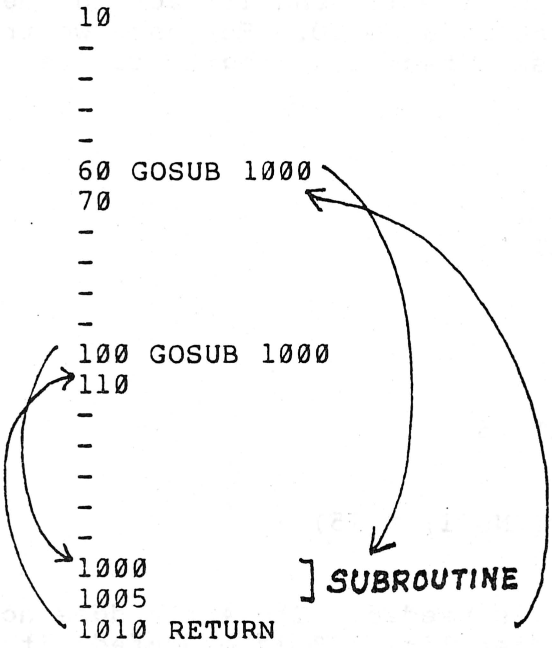
As our programming skills have increased, so have the lengths of our solutions. Complex programs often call for specialized techniques. In some programs there is a specific task that is performed repeatedly within the main program and is accessed from many places. It might be drawing a picture, performing a mathematical function, reading data, generating random numbers. With such a configuration it is possible to create a small program within the main program. This specialized portion is called a SUBROUTINE. The subroutine is accessed by a GOSUB statement. In this statement GOSUB is followed by the line number that tells the Apple where the subroutine begins.

```
60 GOSUB 1000
```

GOSUB is an unconditional transfer. The program jumps to line 1000 and proceeds to execute the statements of the subroutine. The subroutine continues until a RETURN statement. This indicates it is finished and causes the Apple to return to the line after the GOSUB statement.

```
1010 RETURN
```

Within the skeleton framework of a program the GOSUB and RETURN statements work like this:



```
10
-
-
-
-
60 GOSUB 1000
70
-
-
-
-
100 GOSUB 1000
110
-
-
-
-
-
1000
1005
1010 RETURN
```

When statement 60 is executed, the program jumps to line 1000 and executes the subroutine. At 1010, the program jumps back to line 70. When 100 is executed, once again the program transfers to the subroutine start, 1000. This time RETURN sends us back to line 110.

We would like to return to a previous program to demonstrate the subroutine use. Below is our program for drawing the HAPPY FACE.

```
10 GR
20 COLOR = 12
30 FOR X = 3 TO 20
40 HLIN 3, 15 AT X
50 NEXT X
60 COLOR = 11
70 PLOT 7,9
80 PLOT 11,9
85 COLOR = 3
90 HLIN 7, 11 AT 15
100 PLOT 6,14
110 PLOT 12,14
```

Suppose we wanted to draw the face, eyes, and mouth each in a random color. One approach is to retype lines 20, 60, and 85 to the same statement.

```
COLOR = INT(RND(1) * 15) + 1
```

Since this is a part of the program that is repeated several times (and since typing this line is a chore for a beginning typist) we can use the subroutine idea. At each of these lines -- 20, 60, and 85 -- we will send the program to a subroutine at line 200.

```
GOSUB 200
```

Line 200 will be a random color generator.

```
200 COLOR = INT(RND(1) * 15) + 1
```

This accomplished, a RETURN statement will send it back to the appropriate location. This could not be done by a GO TO. For once we transferred to 200, there would be no way to send back the program to the correct spot. The program now becomes:

```
10 GR
20 GOSUB 200
30 FOR X = 3 TO 20
40 HLIN 1, 15 AT X
50 NEXT X
60 GOSUB 200
70 PLOT 7,9
80 PLOT 11,9
85 GOSUB 200
90 HLIN 7, 11 AT 15
100 PLOT 6,14
110 PLOT 12,14
200 COLOR = INT(RND(1) * 15) + 1
205 RETURN
```

One addition to this program is needed. The Apple does not understand the purpose of a subroutine. After line 110 is executed, it will proceed to the next statement 200 and begin to execute the subroutine. Be careful to stop the program at the end of the main program, so that the subroutine is not done by mistake at the end.

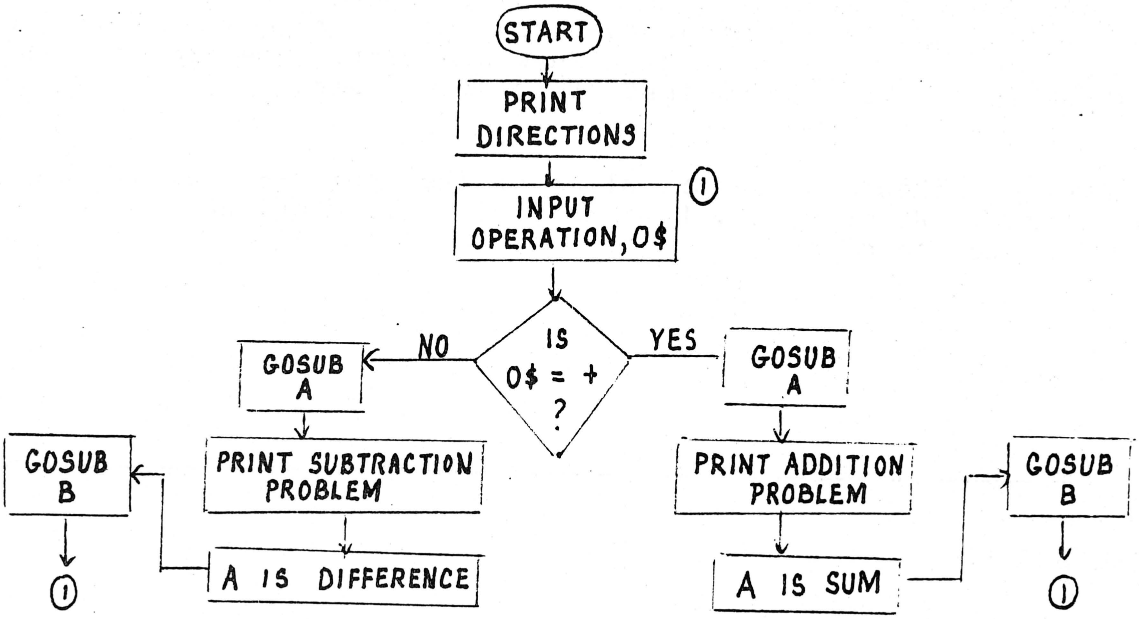
```
120 END
```

One more example will demonstrate the use of subroutines in an "options" program.

We like to keep our sixth graders on their toes with math facts. We call it MENTAL MATH (M & M) and it requires no pencil and paper. It's quick recall, high anxiety. Write a program to generate random problems of the type: 6 + 8, -12 + 8, 4 - 8. Allow the user the option of selecting the operation -- addition or subtraction. Give him/her a quick feedback. This is really two programs in one -- there is an addition and a subtraction portion. But within each of these the same tasks are repeated. Those tasks will become subroutines. Here are our two.

A.) PICK RANDOM NUMBERS
B.) INPUT ANSWER AND CORRECT

In this program we will create two subroutines. Each will be accessed for one of the two main branches.


START
PRINT DIRECTIONS
INPUT OPERATION, O$
①
IS O$ = + ?
NO
YES
GOSUB A
GOSUB A
PRINT SUBTRACTION PROBLEM
PRINT ADDITION PROBLEM
A IS DIFFERENCE
A IS SUM
GOSUB B
GOSUB B
①
①

```
10 PRINT "MENTAL MATH"
11 PRINT
12 PRINT "THE APPLE WILL GIVE YOU"
13 PRINT "A QUICK PROBLEM TO PERFORM"
14 PRINT "TYPE IN THE CORRECT ANSWER"
15 PRINT
16 PRINT "YOU CAN CHOOSE THE OPERATION"
17 INPUT "ADDITION (+) OR SUBTRACTION (-): ";OS
18 IF OS = "+" THEN 100
20 GOSUB 2000
30 PRINT C;"-"D;"="
40 LET A = C-D
50 GOSUB 2000
60 GO TO 16
100 GOSUB 1000
110 PRINT C;"+";D;"="
120 LET A = C + D
130 GOSUB 2000
140 GO TO 16
```

```
1000 LET C = INT(RND(1) * 20)-10
1010 LET D = INT(RND(1) * 20)-10
1020 RETURN

2000 INPUT GUESS
2010 IF GUESS = A THEN 2050
2020 PRINT "INCORRECT!"
2030 PRINT "THE CORRECT ANSWER IS ";A
2040 GO TO 2060
2050 PRINT "GOOD JOB.  KEEP IT UP"
2060 RETURN
```

In the program, lines 30-40 and 110-120 provide for the different operations. Otherwise the processes are the same. The subroutines allow for joining these programs at those tasks which are the same.

GOSUB adds up to a great way of handling repetitive portions of a program. The ability to return anywhere within the program, using the RETURN statement, makes subroutines the right answer.

We have only shown you a thin slice of the Apple -- what it can do with you in control. We sincerely hope that you will continue, slice by slice, until you get to the core!

EXERCISES 6.4

1. Which of the following are acceptable forms of subroutine commands?

A.) GOSUB 2000
B.) GO TO 2000
C.) GOSUB LINE 2000
D.) RETURN TO 110
E.) GO BACK
F.) RETURN 110
G.) RETURN

2. Trace the following program. A correct tracing will please Euclid immensely.

```
10 READ A,B,C
15 DATA 5,10,15,2,3,6
20 LET D = A
30 LET E = B
40 GOSUB 200
50 LET D = C
60 LET E = G
70 GOSUB 200
80 PRINT A,B,C
90 PRINT G
100 END
```

```
200 LET R = E - D * INT(E/D)
210 IF R = O THEN 250
220 LET E = D
230 LET D = E
240 GO TO 200
250 LET G = D
260 RETURN
```

3. Our Language Arts teachers have discovered a wondrous machine called a TACHISTOSCOPE -- a controlled-speed reader. But our Apple can also do the job. In your program, input a sentence as a single string. Then print out one character at a time. Every time a space is encountered, use a delay subroutine before continuing. You might even provide a means of changing speeds.

4. Our first decision box was translated in the program to determine if a temperature was above, below, or freezing. Rewrite that program so that it not only determines the correct response, but also draws a thermometer demonstrating the temperature. Use HI-RES graphics to draw the thermometer outline and then draw the mercury line in red. Use GOSUBs to help you.

A LITTLE BIT MORE SECTION 6

MORE IF'S

In our discussion earlier we translated the decision box as the IF-THEN statement. It has as its form:

IF(CONDITION) THEN (STATEMENT NUMBER)

There are other members of the IF-tHEN family. In fact, it is possible to follow the THEN with almost any command statement.

Look at the example below:

```
5 REM: PROGRAM TELLS IF CELSIUS TEMPERATURE IS
       ABOVE/BELOW ZERO
10 READ X
20 DATA 20,-14,9,-3
30 IF X < 0 THEN PRINT X;"IS BELOW FREEZING": GO TO 10
40 PRINT X;"IS ABOVE FREEZING": GO TO 10
```

In line 30 you see an IF-THEN statement. The rules for the THEN PRINT statement formatting is the same for any regular PRINT statement. IF-THEN PRINT only prints if the condition is true. If the condition is not true, the computer will not execute anything after the THEN. It will go on to the next numbered statement. In our example, when X is -14, then the PRINT statement will output:

```
-14 IS BELOW FREEZING.
```

When X is 20, the condition is false and line 40 prints:

```
20 IS ABOVE FREEZING.
```

Note the use of the multiple statements with this IF-THEN PRINT. In our program, we must follow the affirmative condition printing with a GO TO statement. This avoids both the YES and NO paths eventually leading to line 40. When using this special form of the IF-THEN statement, it will often be necessary to use the multiple statement line.

Not only can we use a PRINT statement, but other commands can be included. Below is an example of the use of INPUT.

```
10 INPUT "DO YOU WANT TO ADD TWO NUMBERS? ";AS
20 IF A$ = "YES" THEN INPUT A,B: GO TO 40
30 END
40 PRINT A;" + ";B;" = ";A + B
50 GO TO 10
```

We talked earlier about the use of a flag -- YES or NO to end a program. In this program, we have been able to include the test with the continuation of the program. If the A$ input value is YES, the condition of line 20 is true and the program proceeds to ask for the addends to be inputted. If the A$ input is anything besides YES the program stops. Again the use of the GO TO statement at the end of line 20 is essential to prevent 30 from being the next statement executed.

The IF-THEN statement family accepts any of these forms:

```
IF...THEN(GO TO)
IF...THEN PRINT
IF...THEN LET
IF...THEN INPUT
IF...THEN READ
IF...THEN END
IF...THEN GOSUB
IF...THEN RETURN
```

Any of these examples can save work and excess statements, but be sure to trace through your program to avoid incorrect branching. Don't leave any IF's to chance.

REVIEW EXERCISES

CHAPTER SIX

1. Write a program to simulate the ancient story of the chessboard. As the story goes, a chieftan would answer the riddle of the sheik for a price. He asked that the sheik pay him in grains of wheat placed on a chessboard as follows: one grain on the first, two grains on second, four on the third, and so forth, doubling the number of grains each time. The sheik agreed. How many grains did the reward amount to? Write the program which prints the total amount and then the amount on each square -- 1 through 64.

2. Mr. Plum, the history teacher, has a problem getting dressed in the morning. He is color blind and his wardrobe reflects it. He has three pairs of pants -- gray, blue, and tan. He has four shirts -- blue, white, blue-striped, and yellow. He has two ties -- dark blue and brown. Write a program which stores his wardrobe as three string arrays. Then randomly print out ten possible outfits. The output should look like this:

```
PANTS   SHIRT   TIE
BLUE    BLUE    DARK BLUE
TAN     BLUE    BROWN
```

3. In the Section 6.2 exercises, you wrote a program to act like the infamous "little black book". Modify the program to do the following. Allow the user to access the information according to category. The program should allow a key word in a category to be inputted like BLUE (for eyes). All the information on males with blue eyes should be then outputted.

4. Write a special Scrabble dictionary list. Store the letters of the alphabet in an array. Select four random numbers and print out the four letter "word" created. Do this twenty times. How many of the combinations are "real" words and how many are "Scrabble" player words?

5. Write a program to read in two fractions and find their sum. Use subroutines to organize the word. One subroutine should be for finding the least common denominator, one for getting equivalent fractions with the LCD, and one should be for reducing the final sum to lowest terms.